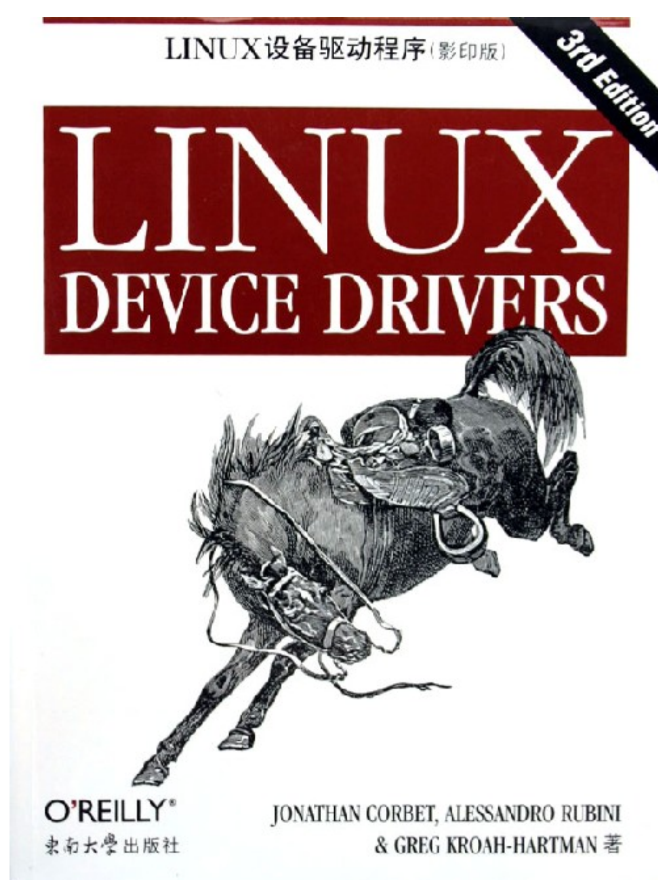
Linux 设备驱动 Edition 3

By

Jonathan Corbet,

Alessandro Rubini,

Greg Kroah-Hartman



目 录

第 1 章 设备驱动简介

以 Linux 为代表的自由操作系统的很多优点之一, 是它们的内部是开放给所有人看的. 操作系统, 曾经是一个隐藏的神秘的地方, 它的代码只局限于少数的程序员, 现在已准备好让任何具备必要技能的人来检查, 理解以及修改. Linux 已经帮助使操作系统民主化. Linux 内核保留有大量的复杂的代码, 但是, 那些想要成为内核 hacker 的人需要一个入口点, 这样他们可以进入代码中, 不会被代码的复杂性压倒. 通常, 设备驱动提供了这样的门路.

驱动程序在 Linux 内核里扮演着特殊的角色. 它们是截然不同的"黑盒子", 使硬件的特殊的一部分响应定义好的内部编程接口. 它们完全隐藏了设备工作的细节. 用户的活动通过一套标准化的调用来进行, 这些调用与特别的驱动是独立的; 设备驱动的角色就是将这些调用映射到作用于实际硬件的和设备相关的操作上. 这个编程接口是这样, 驱动可以与内核的其他部分分开建立, 并在需要的时候在运行时"插入". 这种模块化使得 Linux 驱动易写, 以致于目前有几百个驱动可用.

编写 Linux 设备驱动有许多理由让人感兴趣. 可用的新硬件出现的速率(以及陈旧的速率)就确保了驱动编写者在可见的将来内是忙碌的. 个别人可能需要了解驱动以便存取一个他们感兴趣的特殊设备. 硬件供应商, 通过为他们的产品开发 Linux 驱动, 可以给他们的潜在市场增加大量的正在扩张的 Linux 用户基数. 还有 Linux 系统的开放源码性质意味着如果驱动编写者愿意, 驱动源码能够快速地散布到几百万用户.

本书指导你如何编写你自己的驱动, 以及如何利用内核相关的部分. 我们采用一种设备-独立的方法; 编程技术和接口, 在任何可能的时候, 不会捆绑到任何特定的设备. 每一个驱动都是不同的; 作为一个驱动编写者, 你需要深入理解你的特定设备. 但是大部分的原则和基本技术对所有驱动都是一样的. 本书无法教你关于你的设备的东西, 但是它给予你所需要的使你的设备运行起来的背景知识的指导.

在你学习编写驱动时, 你通常会发现大量有关 Linux 内核的东西. 这也许会帮助你理解你的机器是如何工作的, 以及为什么事情不是如你所愿的快, 或者不是如你所要的进行. 我们会逐步介绍新概念, 由非常简单的驱动开始并建立它们; 每一个新概念都伴有例子代码, 这样的代码不需要特别的硬件来测试.

本章不会真正进入编写代码. 但是, 我们介绍一些 Linux 内核的背景概念, 这样在以后我们动手编程时, 你会感到乐于知道这些.

1.1. 驱动程序的角色

作为一个程序员, 你能够对你的驱动作出你自己的选择, 并且在所需的编程时间和结果的灵活性之间, 选择一个可接受的平衡. 尽管说一个驱动是"灵活"的, 听起来有些奇怪, 但是我们喜欢这个字眼, 因为它强调了一个驱动程序的角色是提供机制, 而不是策略.

机制和策略的区分是其中一个在 Unix 设计背后的最好观念. 大部分的编程问题其实可以划分为 2 部分:" 提供什么能力"(机制) 和 "如何使用这些能力"(策略). 如果这两方面由程序的不同部分来表达, 或者甚至由不同的程序共同表达, 软件包是非常容易开发和适应特殊的需求.

例如, 图形显示的 Unix 管理划分为 X 服务器, 它理解硬件以及提供了统一的接口给用户程序, 还有窗口和会话管理器, 它实现了一个特别的策略, 而对硬件一无所知. 人们可以在不同的硬件上使用相同的窗口管理器, 而且不同的用户可以在同一台工作站上运行不同的配置. 甚至完全不同的桌面环境, 例如 KDE 和 GNOME, 可以在同一系统中共存. 另一个例子是 TCP/IP 网络的分层结构: 操作系统提供 socket 抽象层, 它对要传送的数据

而言不实现策略，而不同的服务器负责各种服务(以及它们的相关策略). 而且，一个服务器，例如 ftpd 提供文件传输机制，同时用户可以使用任何他们喜欢的客户端；无论命令行还是图形客户端都存在，并且任何人都能编写一个新的用户接口来传输文件.

在驱动相关的地方，机制和策略之间的同样的区分都适用. 软驱驱动是不含策略的--它的角色仅仅是将磁盘表现为一个数据块的连续阵列. 系统的更高级部分提供了策略，例如谁可以存取软驱驱动，这个软驱是直接存取还是要通过一个文件系统，以及用户是否可以加载文件系统到这个软驱. 因为不同的环境常常需要不同的使用硬件的方式，尽可能对策略透明是非常重要的.

在编写驱动时，程序员应当特别注意这个基础的概念：编写内核代码来存取硬件，但是不能强加特别的策略给用户，因为不同的用户有不同的需求. 驱动应当做到使硬件可用，将所有关于如何使用硬件的事情留给应用程序. 一个驱动，这样，就是灵活的，如果它提供了对硬件能力的存取，没有增加约束. 然而，有时必须作出一些策略的决定. 例如，一个数字 I/O 驱动也许只提供对硬件的字符存取，以便避免额外的代码处理单个位.

你也可以从不同的角度看你的驱动：它是一个存在于应用程序和实际设备间的软件层. 驱动的这种特权的角色允许驱动程序员严密地选择设备应该如何表现：不同的驱动可以提供不同的能力，甚至是同一个设备. 实际的驱动设计应当是在许多不同考虑中的平衡. 例如，一个单个设备可能由不同的程序并发使用，驱动程序员有完全的自由来决定如何处理并发性. 你能在设备上实现内存映射而不依赖它的硬件能力，或者你能提供一个用户库来帮助应用程序员在可用的原语之上实现新策略，等等. 一个主要的考虑是在展现给用户尽可能多的选项，和你不得不花费的编写驱动的时间之间做出平衡，还有需要保持事情简单以避免错误潜入.

对策略透明的驱动有一些典型的特征. 这些包括支持同步和异步操作，可以多次打开的能力，利用硬件全部能力，没有软件层来"简化事情"或者提供策略相关的操作. 这样的驱动不但对他们的最终用户好用，而且证明也是易写易维护的. 成为策略透明的实际是一个共同的目标，对软件设计者来说.

许多设备驱动，确实，是与用户程序一起发行的，以便帮助配置和存取目标设备. 这些程序包括简单的工具到完全的图形应用. 例子包括 tunelp 程序，它调整并口打印机驱动如何操作，还有图形的 cardctl 工具，它是 PCMCIA 驱动包的一部分. 经常会提供一个客户库，它提供了不需要驱动自身实现的功能.

本书的范围是内核，因此我们尽力不涉及策略问题，应用程序，以及支持库. 有时我们谈论不同的策略以及如何支持他们，但是我们不会进入太多有关使用设备的程序的细节，或者是他们强加的策略的细节. 但是，你应当理解，用户程序是一个软件包的构成部分，并且就算是对策略透明的软件包在发行时也会带有配置文件，来对底层的机制应用缺省的动作.

1.2. 划分内核

在 Unix 系统中，几个并发的进程专注于不同的任务. 每个进程请求系统资源，象计算能力，内存，网络连接，或者一些别的资源. 内核是个大块的可执行文件，负责处理所有这样的请求. 尽管不同内核任务间的区别常常不是能清楚划分，内核的角色可以划分(如同图内核的划分)成下列几个部分:

进程管理

内核负责创建和销毁进程, 并处理它们与外部世界的联系(输入和输出). 不同进程间通讯(通过信号, 管道, 或者进程间通讯原语)对整个系统功能来说是基本的, 也由内核处理. 另外, 调度器, 控制进程如何共享 CPU, 是进程管理的一部分. 更通常地, 内核的进程管理活动实现了多个进程在一个单个或者几个 CPU 之上的抽象.

内存管理

计算机的内存是主要的资源, 处理它所用的策略对系统性能是至关重要的. 内核为所有进程的每一个都在有限的可用资源上建立了一个虚拟地址空间. 内核的不同部分与内存管理子系统通过一套函数调用交互, 从简单的 malloc/free 对到更多更复杂的功能.

文件系统

Unix 在很大程度上基于文件系统的概念; 几乎 Unix 中的任何东西都可看作一个文件. 内核在非结构化的硬件之上建立了一个结构化的文件系统, 结果是文件的抽象非常多地在整个系统中应用. 另外, Linux 支持多个文件系统类型, 就是说, 物理介质上不同的数据组织方式. 例如, 磁盘可被格式化成标准 Linux 的 ext3 文件系统, 普遍使用的 FAT 文件系统, 或者其他几个文件系统.

设备控制

几乎每个系统操作最终都映射到一个物理设备上. 除了处理器, 内存和非常少的别的实体之外, 全部中的任何设备控制操作都由特定于要寻址的设备相关的代码来进行. 这些代码称为设备驱动. 内核中必须嵌入系统中出现的每个外设的驱动, 从硬盘驱动到键盘和磁带驱动器. 内核功能的这个方面是本书中的我们主要感兴趣的地方.

网络

网络必须由操作系统来管理, 因为大部分网络操作不是特定于某一个进程: 进入系统的报文是异步事件. 报文在某一个进程接手之前必须被收集, 识别, 分发. 系统负责在程序和网络接口之间递送数据报文, 它必须根据程序的网络活动来控制程序的执行. 另外, 所有的路由和地址解析问题都在内核中实现.

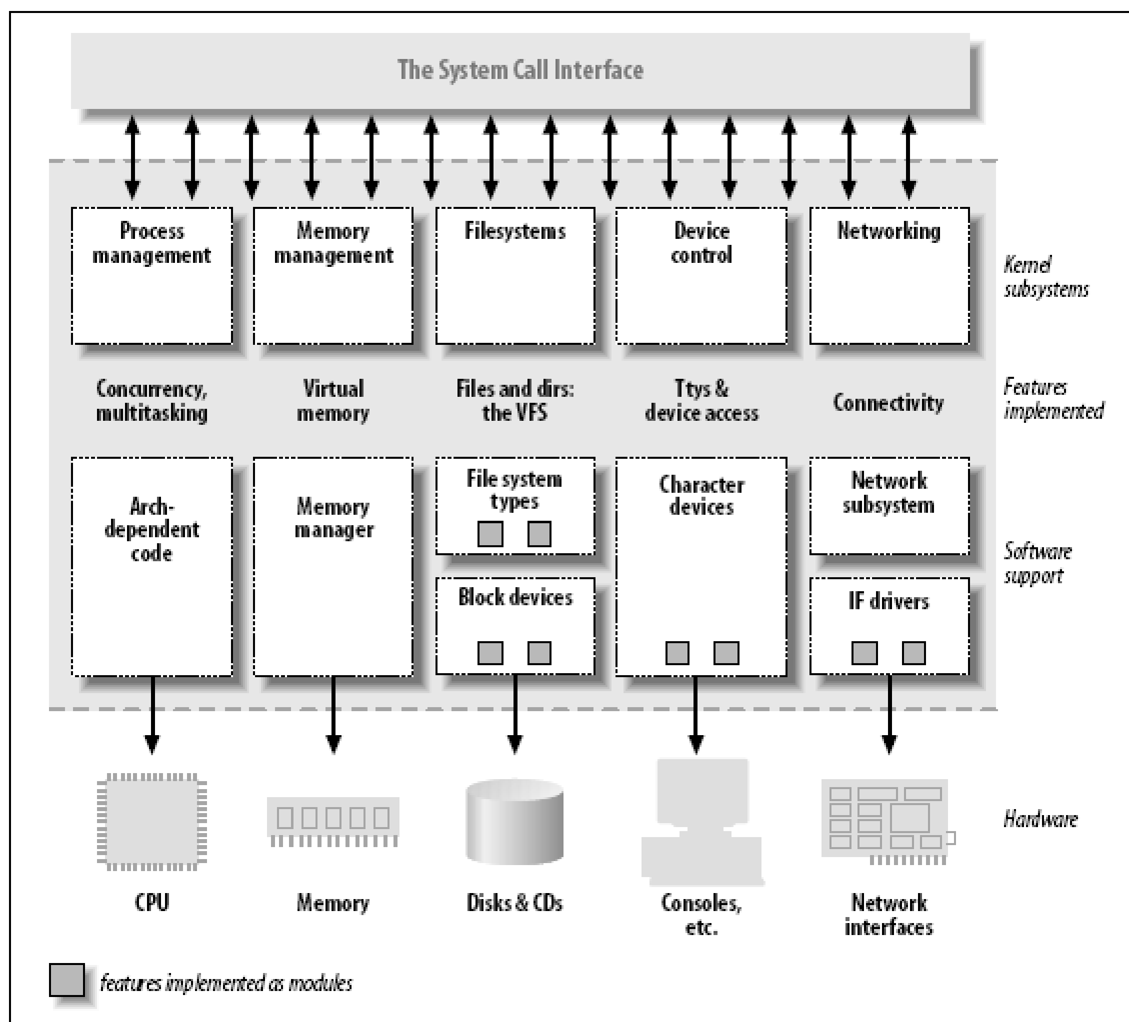
1.2.1. 可加载模块

Linux 的众多优良特性之一就是可以在运行时扩展由内核提供的特性的能力. 这意味着你可以在系统正在运行着的时候增加内核的功能(也可以去除).

每块可以在运行时添加到内核的代码, 被称为一个模块. Linux 内核提供了对许多模块类型的支持, 包括但不限于, 设备驱动. 每个模块由目标代码组成(没有连接成一个完整可执行文件), 可以动态连接到运行中的内核中, 通过 insmod 程序, 以及通过 rmmod 程序去连接.

图 内核的划分 表示了负责特定任务的不同类别的模块，一个模块是根据它提供的功能来说它属于一个特别类别的. 图 内核的划分 中模块的安排涵盖了最重要的类别，但是远未完整，因为在 Linux 中越来越多的功能被模块化了.

图 1.1. 内核的划分



1.3. 设备和模块的分类

以 Linux 的方式看待设备可区分为 3 种基本设备类型. 每个模块常常实现 3 种类型中的 1 种，因此可分类成字符模块，块模块，或者一个网络模块. 这种将模块分成不同类型或类别的方法并非是固定不变的；程序员可以选择建立在一个大块代码中实现了不同驱动的巨大模块. 但是，好的程序员，常常创建一个不同的模块给每个它们实现的新功能，因为分解是可伸缩性和可扩张性的关键因素.

3 类驱动如下：

字符设备

一个字符(char) 设备是一种可以当作一个字节流来存取的设备(如同一个文件)；一个字符驱动负责实现这种行为. 这样的驱动常常至少实现 open，close，

read, 和 write 系统调用. 文本控制台(/dev/console)和串口(/dev/ttyS0 及其友)是字符设备的例子, 因为它们很好地展现了流的抽象. 字符设备通过文件系统结点来存取, 例如 /dev/tty1 和 /dev/lp0. 在一个字符设备和一个普通文件之间唯一有关的不同就是, 你经常可以在普通文件中移来移去, 但是大部分字符设备仅仅是数据通道, 你只能顺序存取.然而, 存在看起来象数据区的字符设备, 你可以在里面移来移去. 例如, frame grabber 经常这样, 应用程序可以使用 mmap 或者 lseek 存取整个要求的图像.

块设备

如同字符设备, 块设备通过位于 /dev 目录的文件系统结点来存取. 一个块设备(例如一个磁盘)应该是可以驻有一个文件系统的. 在大部分的 Unix 系统, 一个块设备只能处理这样的 I/O 操作, 传送一个或多个长度经常是 512 字节(或一个更大的 2 的幂的数)的整块. Linux, 相反, 允许应用程序读写一个块设备象一个字符设备一样 -- 它允许一次传送任意数目的字节. 结果就是, 块和字符设备的区别仅仅在内核在内部管理数据的方式上, 并且因此在内核/驱动的软件接口上不同. 如同一个字符设备, 每个块设备都通过一个文件系统结点被存取的, 它们之间的区别对用户是透明的. 块驱动和字符驱动相比, 与内核的接口完全不同.

网络接口

任何网络事务都通过一个接口来进行, 就是说, 一个能够与其他主机交换数据的设备. 通常, 一个接口是一个硬件设备, 但是它也可能是一个纯粹的软件设备, 比如环回接口. 一个网络接口负责发送和接收数据报文, 在内核网络子系统的驱动下, 不必知道单个事务是如何映射到实际的被发送的报文上的. 很多网络连接(特别那些使用 TCP 的)是面向流的, 但是网络设备却常常设计成处理报文的发送和接收. 一个网络驱动对单个连接一无所知; 它只处理报文.

既然不是一个面向流的设备, 一个网络接口就不象 /dev/tty1 那么容易映射到文件系统的一个结点上. Unix 提供的对接口的存取的方式仍然是通过分配一个名子给它们(例如 eth0), 但是这个名子在文件系统中没有对应的入口. 内核与网络设备驱动间的通讯与字符和块设备驱动所用的完全不同. 不用 read 和 write, 内核调用和报文传递相关的函数.

有其他的划分驱动模块的方式, 与上面的设备类型是正交的. 通常, 某些类型的驱动与给定类型设备的其他层的内核支持函数一起工作. 例如, 你可以说 USB 模块, 串口模块, SCSI 模块, 等等. 每个 USB 设备由一个 USB 模块驱动, 与 USB 子系统一起工作, 但是设备自身在系统中表现为一个字符设备(比如一个 USB 串口), 一个块设备(一个 USB 内存读卡器), 或者一个网络设备(一个 USB 以太网接口).

另外的设备驱动类别近来已经添加到内核中, 包括 FireWire 驱动和 I2O 驱动. 以它们处理 USB 和 SCSI 驱动相同的方式, 内核开发者集合了类别范围内的特性, 并把它们输出给驱动实现者, 以避免重复工作和 bug, 因此简化和加强了编写类似驱动的过程.

在设备驱动之外, 别的功能, 不论硬件和软件, 在内核中都是模块化的. 一个普通的例子是文件系统. 一个文件系统类型决定了在块设备上信息是如何组织的, 以便能表示一棵目录与文件的树. 这样的实体不是设备驱动, 因为没有明确的设备与信息摆放方式相联系;

文件系统类型却是一种软件驱动，因为它将低级数据结构映射为高级的数据结构．文件系统决定一个文件名多长，以及在一个目录入口中存储每个文件的什么信息．文件系统模块必须实现最低级的系统调用，来存取目录和文件，通过映射文件名和路径(以及其他信息，例如存取模式)到保存在数据块中的数据结构．这样的一个接口是完全与数据被传送来去磁盘(或其他介质)相互独立，这个传送是由一个块设备驱动完成的．

如果你考虑一个 Unix 系统是多么依赖下面的文件系统，你会认识到这样的一个软件概念对系统操作是至关重要的．解码文件系统信息的能力处于内核层级中最低级，并且是最重要的；甚至如果你为你的新 CD-ROM 编写块驱动，如果你对上面的数据不能运行 ls 或者 cp 就毫无用处．Linux 支持一个文件系统模块的概念，其软件接口声明了不同操作，可以在一个文件系统节点，目录，文件和超级块上进行操作．对一个程序员来说，居然需要编写一个文件系统模块是非常不常见的，因为官方内核已经包含了大部分重要的文件系统类型的代码．

1.4. 安全问题

安全是当今重要性不断增长的关注点．我们将讨论安全相关的问题，在它们在本书中出现时．有几个通用的概念，却值得现在提一下．

系统中任何安全检查都由内核代码强加上去．如果内核有安全漏洞，系统作为一个整体就有漏洞．在官方的内核发布里，只有一个有授权的用户可以加载模块；系统调用 init_module 检查调用进程是否是有权加载模块到内核里．因此，当运行一个官方内核时，只有超级用户[1][1]或者一个成功获得特权的入侵者，才可以利用特权代码的能力．

在可能时，驱动编写者应当避免将安全策略编到他们的代码中．安全是一个策略问题，最好在内核高层来处理，在系统管理员的控制下．但是，常有例外．

作为一个设备驱动编写者，你应当知道在什么情形下，某些类型的设备存取可能反面地影响系统作为一个整体，并且应当提供足够地控制．例如，会影响全局资源的设备操作(例如设置一条中断线)，可能会损坏硬件(例如，加载固件)，或者它可能会影响其他用户(例如设置一个磁带驱动的缺省的块大小)，常常是只对有足够授权的用户，并且这种检查必须由驱动自身进行．

驱动编写者也必须要小心，当然，来避免引入安全 bug．C 编程语言使得易于犯下几类的错误．例如，许多现今的安全问题是由于缓冲区覆盖引起，它是由于程序员忘记检查有多少数据写入缓冲区，数据在缓冲区结尾之外结束，因此覆盖了无关的数据．这样的错误可能会危及整个系统的安全，必须避免．幸运的是，在设备驱动上下文中避免这样的错误经常是相对容易的，这里对用户的接口经过精细定义并被高度地控制．

一些其他的通用的安全观念也值得牢记．任何从用户进程接收的输入应当以极大的怀疑态度来对待；除非你能核实它，否则不要信任它．小心对待未初始化的内存；从内核获取的任何内存应当清零或者在其对用户进程或设备可用之前进行初始化．否则，可能发生信息泄漏(数据，密码的暴露等等)．如果你的设备解析发送给它的数据，要确保用户不能发送任何能危及系统的东西．最后，考虑一下设备操作的可能后果；如果有特定的操作(例

[1] [1] 从技术上讲，只有具有 CAP_SYS_MODULE 权利的人才可以进行这个操作．我们第 6 章讨论 capabilities ．

如，加载一个适配卡的固件或者格式化一个磁盘），能影响到系统的，这些操作应该完全确定地要限制在授权的用户中.

也要小心，当从第三方接收软件时，特别是与内核有关：因为每个人都可以接触到源码，每个人都可以分拆和重组东西. 尽管你能够信任在你的发布中的预编译的内核，你应当避免运行一个由不能信任的朋友编译的内核 -- 如果你不能作为 root 运行预编译的二进制文件，那么你最好不要运行一个预编译的内核. 例如，一个经过了恶意修改的内核可能会允许任何人加载模块，这样就通过 init_module 开启了一个不想要的后门.

注意，Linux 内核可以编译成不支持任何属于模块的东西，因此关闭了任何模块相关的安全漏洞. 在这种情况下，当然，所有需要的驱动必须直接建立到内核自身内部. 在 2.2 和以后的内核，也可以在系统启动之后，通过 capability 机制来禁止内核模块的加载.

1.5. 版本编号

在深入编程之前，我们应当对 Linux 使用的版本编号方法和本书涉及的版本做些说明.

首先，注意的是在 Linux 系统中使用的每一个软件包有自己的发行版本号，它们之间存在相互依赖性：你需要一个包的特别的版本来运行另外一个包的特别版本. Linux 发布的创建者常常要处理匹配软件包的繁琐问题，这样用户从一个已打包好的发布中安装就不需要处理版本号的问题了. 另外，那些替换和更新系统软件的人，就要自己处理这个问题了. 幸运的是，几乎所有的现代发布支持单个软件包的更新，通过检查软件包之间的依赖性；发布的软件包管理器通常不允许更新，直到满足了依赖性.

为了运行我们在讨论过程中介绍的例子，你除了 2.6 内核要求的之外不需要任何工具的特别版本；任何近期的 Linux 发布都可以用来运行我们的例子. 我们不详述特别的要求，因为你内核源码中的文件 Document/Changes 是这种信息的最好的来源，如果你遇到任何问题.

至于说内核，偶数的内核版本（就是说，2.6.x）是稳定的，用来做通用的发布. 奇数版本（例如 2.7.x），相反，是开发快照并且是非常短暂的；它们的最新版本代表了开发的当前状态，但是会在几天内就过时了.

本书涵盖内核 2.6 版本. 我们的目标是为设备驱动编写者展示 2.6.10 内核的所有可用的特性，这是我们在编写本书时的内核版本. 本书的这一版不涉及内核的其他版本. 你们有人感兴趣的话，本书第 2 版详细涵盖 2.0 到 2.4 版本. 那个版本依然在 htt://lwn.net/Kernel/LDD2 在线获取到.

内核程序员应当明白到 2.6 内核的开发过程的改变. 2.6 系列现在接受之前可能认为对一个"稳定"的内核太大的更改. 在其他的方面，这意味着内核内部编程接口可能改变，因此潜在地会使本书某些部分过时；基于这个原因，伴随着文本的例子代码已知可以在 2.6.10 上运行，但是某些模块没有在之前的版本上编译. 想紧跟内核编程变化的程序员最好加入邮件列表，并且利用列在参考书目中的网站. 也有一个网页在 http://lwn.net/Articls/2.6-kernel-api 上维护，它包含自本书出版以来的 API 改变的信息.

本文不特别地谈论奇数内核版本. 普通用户不会有理由运行开发中的内核. 试验新特性的开发者, 但是, 想运行最新的开发版本. 他们常常不停更新到最新的版本, 来收集 bug 的修正和新的特性实现. 但是注意, 试验中的内核没有任何保障[2][2], 如果你由于一个非当前的奇数版本内核的一个 bug 而引起的问题, 没人可以帮你. 那些运行奇数版本内核的人常常是足够熟练的深入到代码中, 不需要一本教科书, 这也是我们为什么不谈论开发中的内核的另一个原因.

Linux 的另一个特性是它是平台独立的操作系统, 并非仅仅是" PC 克隆体的一种 Unix 克隆 ", 更多的: 它当前支持大约 20 种体系. 本书是尽可能地平台独立, 所有的代码例子至少是在 x86 和 x86-64 平台上测试过. 因为代码已经在 32-bit 和 64-bit 处理器上测试过, 它应当能够在所有其他平台上编译和运行. 如同你可能期望地, 依赖特殊硬件的代码例子不会在所有支持的平台上运行, 但是这个通常在源码里说明了.

1.6. 版权条款

Linux 是以 GNU 通用公共版权(GPL)的版本 2 作为许可的, 它来自自由软件基金的 GNU 项目. GPL 允许任何人重发布, 甚至是销售, GPL 涵盖的产品, 只要接收方对源码能存取并且能够行使同样的权力. 另外, 任何源自使用 GPL 产品的软件产品, 如果它是完全的重新发布, 必须置于 GPL 之下发行.

这样一个许可的主要目的是允许知识的增长, 通过同意每个人去任意修改程序; 同时, 销售软件给公众的人仍然可以做他们的工作. 尽管这是一个简单的目标, 关于 GPL 和它的使用存在着从未结束的讨论. 如果你想阅读这个许可证, 你能够在你的系统中几个地方发现它, 包括你的内核源码树的目录中的 COPYING 文件.

供应商常常询问他们是否可以只发布二进制形式的内核模块. 对这个问题的答案已是有意让它模糊不清. 二进制模块的发布 -- 只要它们依附预已公布的内核接口 -- 至今已是被接受了. 但是内核的版权由许多开发者持有, 并且他们不是全都同意内核模块不是衍生产品. 如果你或者你的雇主想在非自由的许可下发布内核模块, 你真正需要的是和你的法律顾问讨论. 请注意内核开发者不会对于在内核发行之间破坏二进制模块有任何疑虑, 甚至在一个稳定的内核系列之间. 如果它根本上是可能的, 你和你的用户最好以自由软件的方式发行你的模块.

如果你想你的代码进入主流内核, 或者如果你的代码需要对内核的补丁, 你在发行代码时, 必须立刻使用一个 GPL 兼容的许可. 尽管个人使用你的改变不需要强加 GPL, 如果你发布你的代码, 你必须包含你的代码到发布里面 -- 要求你的软件包的人必须被允许任意重建二进制的内容.

至于本书, 大部分的代码是可自由地重新发布, 要么是源码形式, 要么是二进制形式, 我们和 O' Reilly 都不保留任何权利对任何的衍生的工作. 所有的程序都可从 ftp://ftp.ora.com/pub/examples/linux/drivers/ 得到, 详尽的版权条款在相同目录中的 LICENSE 文件里阐述.

---

[2] [2] 注意, 对于偶数版本的内核也不存在保证, 除非你依靠一个同意提供它自己的担保的商业供应商.

1.7. 加入内核开发社团

在你开始为 Linux 内核编写模块时, 你就成为一个开发者大社团的一部分. 在这个社团中, 你不仅会发现有人忙碌于类似工作, 还有一群特别投入的工程师努力使 Linux 成为更好的系统. 这些人可以是帮助, 理念, 以及关键的审查的来源, 以及他们将是你愿意求助的第一类人, 当你在寻找一个新驱动的测试者.

对于 Linux 内核开发者, 中心的汇聚点是 Linux 内核邮件列表. 所有主要的内核开发者, 从 Linus Torvalds 到其他人, 都订阅这个列表. 请注意这个列表不适合心力衰弱的人: 每天或者几天内的书写流量可能多至 200 条消息. 但是, 随这个列表之后的是对那些感兴趣于内核开发的人重要的东西; 它也是一个最高品质的资源, 对那些需要内核开发帮助的人.

为加入 Linux 内核列表, 遵照在 Linux 内核邮件列表 FAQ: http://www.tux.org/lkml 中的指示. 阅读这个 FAQ 的剩余部分, 当你熟悉它时; 那里有大量的有用的信息. Linux 内核开发者都是忙碌的人, 他们更多地愿意帮助那些已经清楚地首先完成了属于自己的那部分工作的人.

1.8. 本书的内容

从这里开始, 我们进入内核编程的世界. 第 2 章介绍了模块化, 解释了内部的秘密以及展示了运行模块的代码. 第 2 章谈论字符驱动以及展示一个基于内存的设备驱动的代码, 出于乐趣对它读写. 使用内存作为设备的硬件基础使得任何人可以不用要求特殊的硬件来运行代码.

调试技术对程序员是必备的工具, 第 4 章介绍它. 对那些想分析当前内核的人同样重要的是并发的管理和竞争情况. 第 5 章关注的是由于并发存取资源而导致的问题, 并且介绍控制并发的 Linux 机制.

在具备了调试和并发管理的能力下, 我们转向字符驱动的高级特性, 例如阻塞操作, selet 的使用, 以及重要的 ioctl 调用; 这是第 6 章的主题.

在处理硬件管理之前, 我们研究多一点内核软件接口: 第 7 章展示了内核中是如何管理时间的, 第 8 章讲解了内存分配.

接下来我们集中到硬件. 第 9 章描述了 I/O 口的管理和设备上的内存缓存; 随后是中断处理, 在 第 10 章. 不幸的是, 不是每个人都能运行这些章节中的例子代码, 因为确实需要某些硬件来测试软件接口中断. 我们尽力保持需要的硬件支持到最小程度, 但是你仍然需要某些硬件, 例如标准并口, 来使用这些章节的例子代码.

第 11 章涉及内核数据类型的使用, 以及编写可移植代码.

本书的第 2 半专注于更高级的主题. 我们从深入硬件内部开始, 特别的, 是特殊外设总线功能. 第 12 章涉及编写 PCI 设备驱动, 第 13 章检验使用 USB 设备的 API.

具有了对外设总线的理解, 我们详细看一下 Linux 设备模型, 这是内核使用的抽象层来描述它管理的硬件和软件资源. 第 14 章是一个自底向上的设备模型框架的考察, 从

kobject 类型开始以及从那里进一步进行. 它涉及设备模型与真实设备的集成; 接下来是利用这些知识来接触如热插拔设备和电源管理等主题.

在第 15 章, 我们转移到 Linux 的内存管理. 这一章显示如何映射系统内存到用户空间(mmap 系统调用), 映射用户内存到内核空间(使用 get_user_pages), 以及如何映射任何一种内存到设备空间(进行 直接内存存取 [DMA] 操作).

我们对内存的理解将对下面两章是有用的, 它们涉及到其他主要的驱动类型. 第 16 章介绍了块驱动, 并展示了与我们到现在为止已遇到过的字符驱动的区别. 第 17 章进入网络驱动的编写. 我们最后是讨论串行驱动(第 18 章)和一个参考书目.

第 2 章 建立和运行模块

时间差不多该开始编程了. 本章介绍所有的关于模块和内核编程的关键概念. 在这几页里, 我们建立并运行一个完整(但是相对地没有什么用处)的模块, 并且查看一些被所有模块共用的基本代码. 开发这样的专门技术对任何类型的模块化的驱动都是重要的基础. 为避免一次抛出太多的概念, 本章只论及模块, 不涉及任何特别的设备类型.

在这里介绍的所有的内核项 (函数, 变量, 头文件, 和宏)在本章的结尾的参考一节里有说明.

2.1. 设置你的测试系统

在本章开始, 我们提供例子模块来演示编程概念. (所有的例子都可从 O'Reilly's 的 FTP 网站上得到, 如第 1 章解释的那样)建立, 加载, 和修改这些例子, 是提高你对驱动如何工作以及如何与内核交互的理解的好方法.

例子模块应该可以在大部分的 2.6.x 内核上运行, 包括那些由发布供应商提供的. 但是, 我们建议你获得一个主流内核, 直接从 kernel.org 的镜像网络, 并把它安装到你的系统中. 供应商的内核可能是主流内核被重重地打了补丁并且和主流内核有分歧; 偶尔, 供应商的补丁可能改变了设备驱动可见的内核 API. 如果你在编写一个必须在特别的发布上运行的驱动, 你当然要在相应的内核上建立和测试. 但是, 处于学习驱动编写的目的, 一个标准内核是最好的.

不管你的内核来源, 建立 2.6.x 的模块需要你有一个配置好并建立好的内核树在你的系统中. 这个要求是从之前内核版本的改变, 之前只要有一套当前版本的头文件就足够了. 2.6 模块针对内核源码树里找到的目标文件连接; 结果是一个更加健壮的模块加载器, 还要求那些目标文件也是可用的. 因此你的第一个商业订单是具备一个内核源码树(或者从 krenel.org 网络或者你的发布者的内核源码包), 建立一个新内核, 并且安装到你的系统. 因为我们稍后会见到的原因, 生活通常是最容易的如果当你建立模块时真正运行目标内核, 尽管这不是需要的.


注意

你应当也考虑一下在哪里进行你的模块试验, 开发和测试. 我们已经尽力使我们的例子模块安全和正确, 但是 bug 的可能性是经常会有的. 内核代码中的错误可能会引起一个用户进程的死亡, 或者偶尔, 瘫痪整个系统. 它们正常地不会导致更严重地后果, 例如磁盘损伤. 然而, 还是建议你进行你的内核试验在一个没有包含你负担不起丢失的数据的系统, 并且没有进行重要的服务. 内核开发者典型地会保留一台"牺牲"系统来测试新的代码.

因此, 如果你还没有一个合适的系统, 带有一个配置好并建立好的源码树在磁盘上, 现在是时候建立了. 我们将等待. 一旦这个任务完成, 你就准备好开始摆布内核模块了.

2.2. Hello World 模块

许多编程书籍从一个 "hello world" 例子开始, 作为一个展示可能的最简单的程序的方法. 本书涉及的是内核模块而不是程序; 因此, 对无耐心的读者, 下面的代码是一个完整的 "hello world"模块:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello, world\n");
        return 0;
}
static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

这个模块定义了两个函数, 一个在模块加载到内核时被调用(hello_init)以及一个在模块被去除时被调用(hello_exit). moudle_init 和 module_exit 这几行使用了特别的内核宏来指出这两个函数的角色. 另一个特别的宏 (MODULE_LICENSE) 是用来告知内核, 该模块带有一个自由的许可证; 没有这样的说明, 在模块加载时内核会抱怨.

printk 函数在 Linux 内核中定义并且对模块可用; 它与标准 C 库函数 printf 的行为相似. 内核需要它自己的打印函数, 因为它靠自己运行, 没有 C 库的帮助. 模块能够调用 printk 是因为, 在 insmod 加载了它之后, 模块被连接到内核并且可存取内核的公用符号 (函数和变量, 下一节详述). 字串 KERN_ALERT 是消息的优先级. [3][3]

我们在此模块中指定了一个高优先级, 因为使用缺省优先级的消息可能不会在任何有用的地方显示, 这依赖于你运行的内核版本, klogd 守护进程的版本, 以及你的配置. 现在你可以忽略这个因素; 我们在第 4 章讲解它.

你可以用 insmod 和 rmmod 工具来测试这个模块. 注意只有超级用户可以加载和卸载模块.

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
 CC [M] /home/ldd3/src/misc-modules/hello.o
 Building modules, stage 2.
 MODPOST
 CC /home/ldd3/src/misc-modules/hello.mod.o
 LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

[3] [3] 优先级只是一个字串, 例如 <1>, 前缀于 printk 格式串之前. 注意在 KERN_ALERT 之后缺少一个逗号; 添加一个逗号在那里是一个普通的讨厌的错误 (幸运的是, 编译器会捕捉到).

请再一次注意，为使上面的操作命令顺序工作，你必须在某个地方有正确配置和建立的内核树，在那里可以找到 makefile (/usr/src/linux-2.6.10，在展示的例子里面). 我们在 "编译和加载" 这一节深入模块建立的细节.

依据你的系统用来递交消息行的机制，你的输出可能不同. 特别地，前面的屏幕输出是来自一个字符控制台；如果你从一个终端模拟器或者在窗口系统中运行 insmod 和 rmmod，你不会在你的屏幕上看到任何东西. 消息进入了其中一个系统日志文件中，例如 /var/log/messages (实际文件名子随 Linux 发布而变化). 内核递交消息的机制在第 4 章描述.

如你能见到的，编写一个模块不是如你想象的困难 -- 至少，在模块没有要求做任何有用的事情时. 困难的部分是理解你的设备，以及如何获得最高性能. 通过本章我们深入模块化内部并且将设备相关的问题留到后续章节.

2.3. 内核模块相比于应用程序

在我们深入之前，有必要强调一下内核模块和应用程序之间的各种不同.

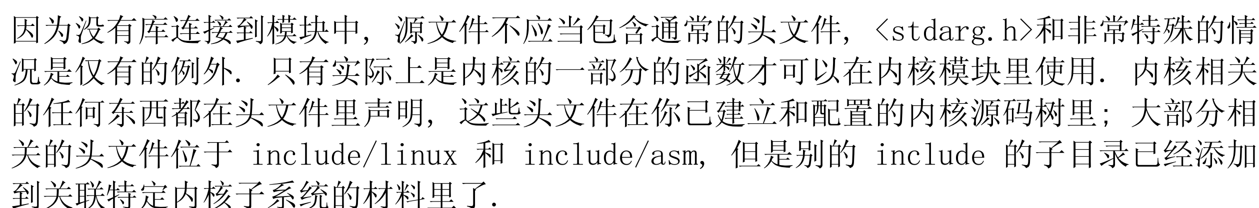
不同于大部分的小的和中型的应用程序从头至尾处理一个单个任务，每个内核模块只注册自己以便来服务将来的请求，并且它的初始化函数立刻终止. 换句话说，模块初始化函数的任务是为以后调用模块的函数做准备；好像是模块说，" 我在这里，这是我能做的."模块的退出函数(例子里是 hello_exit)就在模块被卸载时调用. 它好像告诉内核，"我不再在那里了，不要要求我做任何事了."这种编程的方法类似于事件驱动的编程，但是虽然不是所有的应用程序都是事件驱动的，每个内核模块都是. 另外一个主要的不同，在事件驱动的应用程序和内核代码之间，是退出函数：一个终止的应用程序可以在释放资源方面懒惰，或者完全不做清理工作，但是模块的退出函数必须小心恢复每个由初始化函数建立的东西，否则会保留一些东西直到系统重启.

偶然地，卸载模块的能力是你将最欣赏的模块化的其中一个特色，因为它有助于减少开发时间；你可测试你的新驱动的连续的版本，而不用每次经历漫长的关机/重启周期.

作为一个程序员，你知道一个应用程序可以调用它没有定义的函数：连接阶段使用合适的函数库解决了外部引用. printf 是一个这种可调用的函数并且在 libc 里面定义. 一个模块，在另一方面，只连接到内核，它能够调用的唯一的函数是内核输出的那些；没有库来连接.在 hello.c 中使用的 printk 函数，例如，是在内核中定义的 printf 版本并且输出给模块. 它表现类似于原始的函数，只有几个小的不同，首要的一个是缺乏浮点的支持.

图 连接一个模块到内核 展示了函数调用和函数指针在模块中如何使用来增加新功能到一个运行中的内核.

图 2.1. 连接一个模块到内核



因为没有库连接到模块中，源文件不应当包含通常的头文件，<stdarg.h>和非常特殊的情况是仅有的例外．只有实际上是内核的一部分的函数才可以在内核模块里使用．内核相关的任何东西都在头文件里声明，这些头文件在你已建立和配置的内核源码树里；大部分相关的头文件位于 include/linux 和 include/asm，但是别的 include 的子目录已经添加到关联特定内核子系统的材料里了．

单个内核头文件的作用在书中需要它们的时候进行介绍．

另外一个在内核编程和应用程序编程之间的重要不同是每一个环境是如何处理错误：在应用程序开发中段错误是无害的，一个调试器常常用来追踪错误到源码中的问题，而一个内核错误至少会杀掉当前进程，如果不终止整个系统．我们会在第 4 章看到如何跟踪内核错误．

2.3.1. 用户空间和内核空间

A module runs in kernel space, whereas applications run in user space. This concept is at the base of operating systems theory.
一个模块在内核空间运行，而应用程序在用户空间运行．这个概念是操作系统理论的基础．

操作系统的角色，实际上，是给程序提供一个一致的计算机硬件的视角．另外，操作系统必须承担程序的独立操作和保护对于非授权的资源存取．这一不平凡的任务只有 CPU 增强系统软件对应用程序的保护才有可能．

每种现代处理器都能够加强这种行为．选中的方法是 CPU 自己实现不同的操作形态(或者级别)．这些级别有不同的角色，一些操作在低些级别中不允许；程序代码只能通过有限的几个门从一种级别切换到另一个．Unix 系统设计成利用了这种硬件特性，使用了两个这样的级别．所有当今的处理器至少有两个保护级别，并且某些，例如 x86 家族，有更多级别；当几个级别存在时，使用最高和最低级别．在 Unix 下，内核在最高级运行(也称之为超级模式)，这里任何事情都允许，而应用程序在最低级运行(所谓的用户模式)，这里处理器控制了对硬件的直接存取以及对内存的非法存取．

我们常常提到运行模式作为内核空间和用户空间. 这些术语不仅包含存在于这两个模式中不同特权级别, 还包含有这样的事实, 即每个模式有它自己的内存映射 -- 它自己的地址空间.

Unix 从用户空间转换执行到内核空间, 无论何时一个应用程序发出一个系统调用或者被硬件中断挂起时. 执行系统调用的内核代码在进程的上下文中工作 -- 它代表调用进程并且可以存取该进程的地址空间. 换句话说, 处理中断的代码对进程来说是异步的, 不和任何特别的进程有关.

模块的角色是扩展内核的功能; 模块化的代码在内核空间运行. 经常地一个驱动进行之前提到的两种任务: 模块中一些的函数作为系统调用的一部分执行, 一些负责中断处理.

2.3.2. 内核的并发

内核编程与传统应用程序编程方式很大不同的是并发问题. 大部分应用程序, 多线程的应用程序是一个明显的例外, 典型地是顺序运行的, 从头至尾, 不必要担心其他事情会发生而改变它们的环境. 内核代码没有运行在这样的简单世界中, 即便最简单的内核模块必须在这样的概念下编写, 很多事情可能马上发生.

内核编程中有几个并发的来源. 自然的, Linux 系统运行多个进程, 在同一时间, 不止一个进程能够试图使用你的驱动. 大部分设备能够中断处理器; 中断处理异步运行, 并且可能在你的驱动试图做其他事情的同一时间被调用. 几个软件抽象(例如内核定时器, 第 7 章介绍)也异步运行. 而且, 当然, Linux 可以在对称多处理器系统(SMP)上运行, 结果是你的驱动可能在多个 CPU 上并发执行. 最后, 在 2.6, 内核代码已经是可抢占的了; 这个变化使得即便是单处理器会有许多与多处理器系统同样的并发问题.

结果, Linux 内核代码, 包括驱动代码, 必须是可重入的 -- 它必须能够同时在多个上下文中运行. 数据结构必须小心设计以保持多个执行线程分开, 并且代码必须小心存取共享数据, 避免数据的破坏. 编写处理并发和避免竞争情况(一个不幸的执行顺序导致不希望的行为的情形)的代码需要仔细考虑并可能是微妙的. 正确的并发管理在编写正确的内核代码时是必须的; 由于这个理由, 本书的每一个例子驱动都是考虑了并发下编写的. 用到的技术在我们遇到它们时再讲解; 第 5 章也专门讲述这个问题, 以及并发管理的可用的内核原语.

驱动程序员的一个通常的错误是假定并发不是一个问题, 只要一段特别的代码没有进入睡眠(或者 "阻塞"). 即便在之前的内核(不可抢占), 这种假设在多处理器系统中也不成立. 在 2.6, 内核代码不能(极少)假定它能在一段给定代码上持有处理器. 如果你不考虑并发来编写你的代码, 就极有可能导致严重失效, 以至于非常难于调试.

2.3.3. 当前进程

尽管内核模块不象应用程序一样顺序执行, 内核做的大部分动作是代表一个特定进程的. 内核代码可以引用当前进程, 通过存取全局项 current, 它在 <asm/current.h> 中定义, 它产生一个指针指向结构 task_struct, 在 <linux/sched.h> 定义. current 指针指向当前在运行的进程. 在一个系统调用执行期间, 例如 open 或者 read, 当前进程是发出调用的进程. 内核代码可以通过使用 current 来使用进程特定的信息, 如果它需要这样. 这种技术的一个例子在第 6 章展示.

实际上，current 不真正地是一个全局变量. 支持 SMP 系统的需要强迫内核开发者去开发一种机制，在相关的 CPU 上来找到当前进程. 这种机制也必须快速，因为对 current 的引用非常频繁地发生. 结果就是一个依赖体系的机制，常常，隐藏了一个指向 task_struct 的指针在内核堆栈内. 实现的细节对别的内核子系统保持隐藏，一个设备驱动可以只包含 <linux/sched.h> 并且引用当前进程. 例如，下面的语句打印了当前进程的进程 ID 和命令名称，通过存取结构 task_struct 中的某些字段.

```
printk(KERN_INFO "The process is \"%s\" (pid %i)\n", current->comm, current->pid);
```

存于 current->comm 的命令名称是由当前进程执行的程序文件的基本名称(截短到 15 个字符，如果需要).

2.3.4. 几个别的细节

内核编程与用户空间编程在许多方面不同. 我们将在本书的过程中指出它们，但是有几个基础性的问题，尽管没有保证它们自己有一节内容，也值得一提. 因此，当你深入内核时，下面的事项应当牢记.

应用程序存在于虚拟内存中，有一个非常大的堆栈区. 堆栈，当然，是用来保存函数调用历史以及所有的由当前活跃的函数创建的自动变量. 内核，相反，有一个非常小的堆栈；它可能小到一个，4096 字节的页. 你的函数必须与这个内核空间调用链共享这个堆栈. 因此，声明一个巨大的自动变量从来就不是一个好主意；如果你需要大的结构，你应当在调用时间内动态分配.

常常，当你查看内核 API 时，你会遇到以双下划线(__)开始的函数名. 这样标志的函数名通常是一个低层的接口组件，应当小心使用. 本质上讲，双下划线告诉程序员:" 如果你调用这个函数，确信你知道你在做什么."

内核代码不能做浮点算术. 使能浮点将要求内核在每次进出内核空间的时候保存和恢复浮点处理器的状态 -- 至少，在某些体系上. 在这种情况下，内核代码真的没有必要包含浮点，额外的负担不值得.

2.4. 编译和加载

本章开头的 "hello world" 例子包含了一个简短的建立并加载模块到系统中去的演示. 当然，整个过程比我们目前看到的多. 本节提供了更多细节关于一个模块作者如何将源码转换成内核中的运行的子系统.

2.4.1. 编译模块

第一步，我们需要看一下模块如何必须被建立. 模块的建立过程与用户空间的应用程序的建立过程有显著不同；内核是一个大的，独立的程序，对于它的各个部分如何组合在一起有详细的明确的要求. 建立过程也与以前版本的内核的过程不同；新的建立系统用起来更简单并且产生更正确的结果，但是它看起来与以前非常不同. 内核建立系统是一头负责的野兽，我们就看它一小部分. 在内核源码的 Document/kbuild 目录下发现的文件，任何想理解表面之下的真实情况的人都要阅读一下.

有几个前提，你必须在能建立内核模块前解决．第一个是保证你有版本足够新的编译器，模块工具，以及其他必要工具．在内核文档目录下的文件 Documentation/Changes 一直列出了需要的工具版本；你应当在向前走之前参考一下它．试图建立一个内核(包括它的模块)，用错误的工具版本，可能导致不尽的奇怪的难题．注意，偶尔地，编译器的版本太新可能会引起和太老的版本引起的一样的问题．内核源码对于编译器做了很大的假设，新的发行版本有时会一时地破坏东西．

如果你仍然没有一个内核树在手边，或者还没有配置和建立内核，现在是时间去做了．没有源码树在你的文件系统上，你无法为 2.6 内核建立可加载的模块．实际运行为其而建立的内核也是有帮助的(尽管不是必要的)．

一旦你已建立起所有东西，给你的模块创建一个 makefile 就是直截了当的．实际上，对于本章前面展示的" hello world" 例子，单行就够了：

```
obj-m := hello.o
```

熟悉 make ，但是对 2.6 内核建立系统不熟悉的读者，可能奇怪这个 makefile 如何工作．毕竟上面的这一行不是一个传统的 makefile 的样子．答案，当然，是内核建立系统处理了余下的工作．上面的安排(它利用了由 GNU make 提供的扩展语法)表明有一个模块要从目标文件 hello.o 建立．在从目标文件建立后结果模块命名为 hello.ko．

反之，如果你有一个模块名为 module.ko，是来自 2 个源文件(姑且称之为，file1.c 和 file2.c)，正确的书写应当是：

```
obj-m := module.o
module-objs := file1.o file2.o
```

对于一个象上面展示的要工作的 makefile，它必须在更大的内核建立系统的上下文被调用．如果你的内核源码数位于，假设，你的 ~/kernel-2.6 目录，用来建立你的模块的 make 命令(在包含模块源码和 makefile 的目录下键入)会是：

```
make -C ~/kernel-2.6 M=`pwd` modules
```

这个命令开始是改变它的目录到用 -C 选项提供的目录下(就是说，你的内核源码目录)．它在那里会发现内核的顶层 makefile．这个 M= 选项使 makefile 在试图建立模块目标前，回到你的模块源码目录．这个目标，依次地，是指在 obj-m 变量中发现的模块列表，在我们的例子里设成了 module.o．

键入前面的 make 命令一会儿之后就会感觉烦，所以内核开发者就开发了一种 makefile 方式，使得生活容易些对于那些在内核树之外建立模块的人．这个窍门是如下书写你的 makefile：

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)

 obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
```

```
else

 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 PWD := $(shell pwd)
default:
 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

再一次，我们看到了扩展的 GNU make 语法在起作用. 这个 makefile 在一次典型的建立中要被读 2 次. 当从命令行中调用这个 makefile ，它注意到 KERNELRELEASE 变量没有设置. 它利用这样一个事实来定位内核源码目录，即已安装模块目录中的符号连接指回内核建立树. 如果你实际上没有运行你在为其而建立的内核，你可以在命令行提供一个 KERNELDIR= 选项，设置 KERNELDIR 环境变量，或者重写 makefile 中设置 KERNELDIR 的那一行. 一旦发现内核源码树，makefile 调用 default: 目标，来运行第 2 个 make 命令(在 makefile 里参数化成 $(MAKE))象前面描述过的一样来调用内核建立系统. 在第 2 次读，makefile 设置 obj-m，并且内核的 makefile 文件完成实际的建立模块工作.

这种建立模块的机制你可能感觉笨拙模糊. 一旦你习惯了它，但是，你很可能会欣赏这种已经编排进内核建立系统的能力. 注意，上面的不是一个完整的 makefile；一个真正的 makefile 包含通常的目标类型来清除不要的文件，安装模块等等. 一个完整的例子可以参考例子代码目录的 makefile.

2.4.2. 加载和卸载模块

模块建立之后，下一步是加载到内核. 如我们已指出的，insmod 为你完成这个工作. 这个程序加载模块的代码段和数据段到内核，接着，执行一个类似 ld 的函数，它连接模块中任何未解决的符号连接到内核的符号表上. 但是不象连接器，内核不修改模块的磁盘文件，而是内存内的拷贝. insmod 接收许多命令行选项(详情见 manpage)，它能够安排值给你模块中的参数，在连接到当前内核之前. 因此，如果一个模块正确设计了，它能够在加载时配置；加载时配置比编译时配置给了用户更多的灵活性，有时仍然在用. 加载时配置在本章后面的 "模块参数" 一节讲解.

感兴趣的读者可能想看看内核如何支持 insmod：它依赖一个在 kernel/module.c 中定义的系统调用. 函数 sys_init_module 分配内核内存来存放模块 (这个内存用 vmalloc 分配；看第 8 章的 "vmalloc 和其友")；它接着拷贝模块的代码段到这块内存区，借助内核符号表解决模块中的内核引用，并且调用模块的初始化函数来启动所有东西.

如果你真正看了内核代码，你会发现系统调用的名子以 sys_ 为前缀. 这对所有系统调用都是成立的，并且没有别的函数. 记住这个有助于在源码中查找系统调用.

modprobe 工具值得快速提及一下. modprobe，如同 insmod，加载一个模块到内核. 它的不同在于它会查看要加载的模块，看是否它引用了当前内核没有定义的符号. 如果发现有，modprobe 在定义相关符号的当前模块搜索路径中寻找其他模块. 当 modprobe 找到这些模块(要加载模块需要的)，它也把它们加载到内核. 如果你在这种情况下代替以使用 insmod ，命令会失败，在系统日志文件中留下一条 " unresolved symbols "消息.

如前面提到, 模块可以用 rmmod 工具从内核去除. 注意, 如果内核认为模块还在用(就是说, 一个程序仍然有一个打开文件对应模块输出的设备), 或者内核被配置成不允许模块去除, 模块去除会失败. 可以配置内核允许"强行"去除模块, 甚至在它们看来是忙的. 如果你到了需要这选项的地步, 但是, 事情可能已经错的太严重以至于最好的动作就是重启了.

lsmod 程序生成一个内核中当前加载的模块的列表. 一些其他信息, 例如使用了一个特定模块的其他模块, 也提供了. lsmod 通过读取 /proc/modules 虚拟文件工作. 当前加载的模块的信息也可在位于 /sys/module 的 sysfs 虚拟文件系统找到.

2.4.3. 版本依赖

记住, 你的模块代码一定要为每个它要连接的内核版本重新编译 -- 至少, 在缺乏 modversions 时, 这里不涉及因为它们更多的是给内核发布制作者, 而不是开发者. 模块是紧密结合到一个特殊内核版本的数据结构和函数原型上的; 模块见到的接口可能一个内核版本与另一个有很大差别. 当然, 在开发中的内核更加是这样.

内核不只是认为一个给定模块是针对一个正确的内核版本建立的. 建立过程的其中一步是对一个当前内核树中的文件(称为 vermagic.o)连接你的模块; 这个东东含有相当多的有关要为其建立模块的内核的信息, 包括目标内核版本, 编译器版本, 以及许多重要配置变量的设置. 当尝试加载一个模块, 这些信息被检查与运行内核的兼容性. 如果不匹配, 模块不会加载; 代之的是你见到如下内容:

```
# insmod hello.ko
Error inserting './hello.ko': -1 Invalid module format
```

看一下系统日志文件(/var/log/message 或者任何你的系统被配置来用的)将发现导致模块无法加载特定的问题.

如果你需要编译一个模块给一个特定的内核版本, 你将需要使用这个特定版本的建立系统和源码树. 前面展示过的在例子 makefile 中简单修改 KERNELDIR 变量, 就完成这个动作.

内核接口在各个发行之间常常变化. 如果你编写一个模块想用来在多个内核版本上工作(特别地是如果它必须跨大的发行版本), 你可能只能使用宏定义和 #ifdef 来使你的代码正确建立. 本书的这个版本只关心内核的一个主要版本, 因此不会在我们的例子代码中经常见到版本检查. 但是这种需要确实有时会有. 在这样情况下, 你要利用在 linux/version.h 中发现的定义. 这个头文件, 自动包含在 linux/module.h, 定义了下面的宏定义:

UTS_RELEASE

这个宏定义扩展成字符串, 描述了这个内核树的版本. 例如, "2.6.10".

LINUX_VERSION_CODE

这个宏定义扩展成内核版本的二进制形式，版本号发行号的每个部分用一个字节表示．例如，2.6.10 的编码是 132618 (就是，0x02060a)．[4][4]有了这个信息，你可以(几乎是)容易地决定你在处理的内核版本．

KERNEL_VERSION(major,minor,release)

这个宏定义用来建立一个整型版本编码，从组成一个版本号的单个数字．例如，KERNEL_VERSION(2.6.10) 扩展成 132618．这个宏定义非常有用，当你需要比较当前版本和一个已知的检查点．

大部分的基于内核版本的依赖性可以使用预处理器条件解决，通过利用 KERNEL_VERSION 和 LINUX_VERSION_VODE．版本依赖不应当，但是，用繁多的 #ifdef 条件来搞乱驱动的代码；处理不兼容的最好的方式是把它们限制到特定的头文件．作为一个通用的原则，明显版本(或者平台)依赖的代码应当隐藏在一个低级的宏定义或者函数后面．高层的代码就可以只调用这些函数，而不必关心低层的细节．这样书写的代码易读并且更健壮．

2.4.4. 平台依赖性

每个电脑平台有其自己的特点，内核设计者可以自由使用所有的特性来获得更好的性能．in the target object file ???

不象应用程序开发者，他们必须和预编译的库一起连接他们的代码，依附在参数传递的规定上，内核开发者可以专用某些处理器寄存器给特别的用途，他们确实这样做了．更多的，内核代码可以为一个 CPU 族里的特定处理器优化，以最好地利用目标平台；不象应用程序那样常常以二进制格式发布，一个定制的内核编译可以为一个特定的计算机系列优化．

例如，IA32 (x86) 结构分为几个不同的处理器类型．老式的 80386 处理器仍然被支持(到现在)，尽管它的指令集，以现代的标准看，非常有限．这个体系中更加现代的处理器已经引入了许多新特性，包括进入内核的快速指令，处理器间的加锁，拷贝数据，等等．更新的处理器也可采用 36 位(或者更大)的物理地址，当在适当的模式下，以允许他们寻址超过 4 GB 的物理内存．其他的处理器家族也有类似的改进．内核，依赖不同的配置选项，可以被建立来使用这些附加的特性．

清楚地，如果一个模块与一个给定内核工作，它必须以与内核相同的对目标处理器的理解来建立．再一次，vermagic.o 目标文件登场．当加载一个模块，内核为模块检查特定处理器的配置选项，确认它们匹配运行的内核．如果模块用不同选项编译，它不会加载．

如果你计划为通用的发布编写驱动，你可能很奇怪你怎么可能支持所有这些不同的变体．最好的答案，当然，是发行你的驱动在 GPL 兼容的许可之下，并且贡献它给主流内核．如果没有那样，以源码形式和一套脚本发布你的驱动，以便在用户系统上编译可能是最好的答案．一些供应商已发行了工具来简化这个工作．如果你必须发布你的驱动以二进制形式，你需要查看由你的目标发布所提供的不同的内核，并且为每个提供一个模块版本．要确认考虑到了任何在产生发布后可能发行的勘误内核．接着，要考虑许可权的问题，如同我们

[4] [4] 这允许在稳定版本之间多达 256 个开发版本．

在第 1 章的" 许可条款" 一节中讨论的. 作为一个通用的规则, 以源码形式发布东西是你行于世的易途.

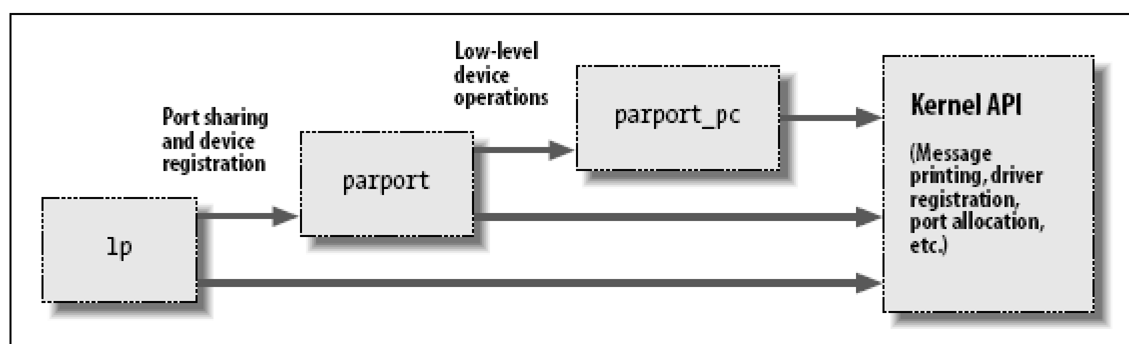
2.5. 内核符号表

我们已经看到 insmod 如何对应共用的内核符号来解决未定义的符号. 表中包含了全局内核项的地址 -- 函数和变量 -- 需要来完成模块化的驱动. 当加载一个模块, 如何由模块输出的符号成为内核符号表的一部分. 通常情况下, 一个模块完成它自己的功能不需要输出如何符号. 你需要输出符号, 但是, 在任何别的模块能得益于使用它们的时候.

新的模块可以用你的模块输出的符号, 你可以堆叠新的模块在其他模块之上. 模块堆叠在主流内核源码中也实现了: msdos 文件系统依赖 fat 模块输出的符号, 某一个输入 USB 设备模块堆叠在 usbcore 和输入模块之上.

模块堆叠在复杂的工程中有用处. 如果一个新的抽象以驱动程序的形式实现, 它可能提供一个特定硬件实现的插入点. 例如, video-for-linux 系列驱动分成一个通用模块, 输出了由特定硬件的低层设备驱动使用的符号. 根据你的设置, 你加载通用的视频模块和你的已安装硬件对应的特定模块. 对并口的支持和众多可连接设备以同样的方式处理, 如同 USB 内核子系统. 在并口子系统的堆叠在图 并口驱动模块的堆叠 中显示; 箭头显示了模块和内核编程接口间的通讯.

图 2.2. 并口驱动模块的堆叠



当使用堆叠的模块时, 熟悉 modprobe 工具是有帮助的. 如我们前面讲的, modprobe 函数很多地方与 insmod 相同, 但是它也加载任何你要加载的模块需要的其他模块. 所以, 一个 modprobe 命令有时可能代替几次使用 insmod(尽管你从当前目录下加载你自己模块仍将需要 insmod, 因为 modprobe 只查找标准的已安装模块目录).

使用堆叠来划分模块成不同层, 这有助于通过简化每一层来缩短开发时间. 这同我们在第 1 章讨论的区分机制和策略是类似的.

linux 内核头文件提供了方便来管理你的符号的可见性, 因此减少了命名空间的污染(将与在内核别处已定义的符号冲突的名子填入命名空间), 并促使了正确的信息隐藏. 如果你的模块需要输出符号给其他模块使用, 应当使用下面的宏定义:

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

上面宏定义的任一个使得给定的符号在模块外可用. _GPL 版本的宏定义只能使符号对 GPL 许可的模块可用. 符号必须在模块文件的全局部分输出, 在任何函数之外, 因为宏定义扩展成一个特殊用途的并被期望是全局存取的变量的声明. 这个变量存储于模块的一个特殊的可执行部分(一个 "ELF 段"), 内核用这个部分在加载时找到模块输出的变量. (感兴趣的读者可以看 <linux/module.h> 获知详情, 尽管并不需要这些细节使东西动起来.)

2.6. 预备知识

我们正在接近去看一些实际的模块代码. 但是首先, 我们需要看一些需要出现在你的模块源码文件中的东西. 内核是一个独特的环境, 它将它的要求强加于要和它接口的代码上.

大部分内核代码包含了许多数量的头文件来获得函数, 数据结构和变量的定义. 我们将在碰到它们时检查这些文件, 但是有几个文件对模块是特殊的, 必须出现在每一个可加载模块中. 因此, 几乎所有模块代码都有下面内容:

```
#include <linux/module.h>
#include <linux/init.h>
```

moudle.h 包含了大量加载模块需要的函数和符号的定义. 你需要 init.h 来指定你的初始化和清理函数, 如我们在上面的 "hello world" 例子里见到的, 这个我们在下一节中再讲. 大部分模块还包含 moudleparam.h, 使得可以在模块加载时传递参数给模块. 我们将很快遇到.

不是严格要求的, 但是你的模块确实应当指定它的代码使用哪个许可. 做到这一点只需包含一行 MODULE_LICENSE:

```
MODULE_LICENSE("GPL");
```

内核认识的特定许可有, "GPL"(适用 GNU 通用公共许可的任何版本), "GPL v2"(只适用 GPL 版本 2), "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL", 和 "Proprietary". 除非你的模块明确标识是在内核认识的一个自由许可下, 否则就假定它是私有的, 内核在模块加载时被"弄污浊"了. 象我们在第 1 章"许可条款"中提到的, 内核开发者不会热心帮助在加载了私有模块后遇到问题的用户.

可以在模块中包含的其他描述性定义有 MODULE_AUTHOR (声明谁编写了模块), MODULE_DESCRIPION(一个人可读的关于模块做什么的声明), MODULE_VERSION (一个代码修订版本号; 看 <linux/module.h> 的注释以便知道创建版本字串使用的惯例), MODULE_ALIAS (模块为人所知的另一个名子), 以及 MODULE_DEVICE_TABLE (来告知用户空间, 模块支持那些设备). 我们会讨论 MODULE_ALIAS 在第 11 章以及 MUDULE_DEVICE_TABLE 在第 12 章.

各种 MODULE_ 声明可以出现在你的源码文件的任何函数之外的地方. 但是, 一个内核代码中相对近期的惯例是把这些声明放在文件末尾.

2.7. 初始化和关停

如已提到的，模块初始化函数注册模块提供的任何功能. 这些功能，我们指的是新功能，可以由应用程序存取的或者一整个驱动或者一个新软件抽象. 实际的初始化函数定义常常如:

```
static int __init initialization_function(void)
{
 /* Initialization code here */
}
module_init(initialization_function);
```

初始化函数应当声明成静态的，因为它们不会在特定文件之外可见; 没有硬性规定这个，然而，因为没有函数能输出给内核其他部分，除非明确请求. 声明中的 __init 标志可能看起来有点怪; 它是一个给内核的暗示，给定的函数只是在初始化使用. 模块加载者在模块加载后会丢掉这个初始化函数，使它的内存可做其他用途. 一个类似的标签 (__initdata) 给只在初始化时用的数据. 使用 __init 和 __initdata 是可选的，但是它带来的麻烦是值得的. 只是要确认不要用在那些在初始化完成后还使用的函数(或者数据结构)上. 你可能还会遇到 __devinit 和 __devinitdata 在内核源码里; 这些只在内核没有配置支持 hotplug 设备时转换成 __init 和 _initdata. 我们会在 14 章谈论 hotplug 支持.

使用 moudle_init 是强制的. 这个宏定义增加了特别的段到模块目标代码中，表明在哪里找到模块的初始化函数. 没有这个定义，你的初始化函数不会被调用.

模块可以注册许多的不同设施，包括不同类型的设备，文件系统，加密转换，以及更多. 对每一个设施，有一个特定的内核函数来完成这个注册. 传给内核注册函数的参数常常是一些数据结构的指针，描述新设施以及要注册的新设施的名子. 数据结构常常包含模块函数指针，模块中的函数就是这样被调用的.

能够注册的项目远远超出第 1 章中提到的设备类型列表. 它们包括，其他的，串口，多样设备，sysfs 入口，/proc 文件，执行域，链路规程. 这些可注册项的大部分都支持不直接和硬件相关的函数，但是处于"软件抽象"区域里. 这些项可以注册，是因为它们以各种方式(例如象 /proc 文件和链路规程)集成在驱动的功能中.

对某些驱动有其他的设施可以注册作为补充，但它们的使用太特别，所以不值得讨论它们. 它们使用堆叠技术，在"内核符号表"一节中讲过. 如果你想深入探求，你可以在内核源码里查找 EXPORT_SYMBOL ，找到由不同驱动提供的入口点. 大部分注册函数以 register_ 做前缀，因此找到它们的另外一个方法是在内核源码里查找 register_ .

2.7.1. 清理函数

每个非试验性的模块也要求有一个清理函数，它注销接口，在模块被去除之前返回所有资源给系统. 这个函数定义为:

```
static void __exit cleanup_function(void)
{
 /* Cleanup code here */
}
module_exit(cleanup_function);
```

清理函数没有返回值，因此它被声明为 void. __exit 修饰符标识这个代码是只用于模块卸载(通过使编译器把它放在特殊的 ELF 段). 如果你的模块直接建立在内核里，或者如果你的内核配置成不允许模块卸载，标识为 __exit 的函数被简单地丢弃. 因为这个原因，一个标识 __exit 的函数只在模块卸载或者系统停止时调用；任何别的使用是错的. 再一次，moudle_exit 声明对于使得内核能够找到你的清理函数是必要的.

如果你的模块没有定义一个清理函数，内核不会允许它被卸载.

2.7.2. 初始化中的错误处理

你必须记住一件事，在注册内核设施时，注册可能失败. 即便最简单的动作常常需要内存分配，分配的内存可能不可用. 因此模块代码必须一直检查返回值，并且确认要求的操作实际上已经成功.

如果在你注册工具时发生任何错误，首先第一的事情是决定模块是否能够无论如何继续初始化它自己. 常常，在一个注册失败后模块可以继续操作，如果需要可以功能降级. 在任何可能的时候，你的模块应当尽力向前，并提供事情失败后具备的能力.

如果证实你的模块在一个特别类型的失败后完全不能加载，你必须取消任何在失败前注册的动作. 内核不保留已经注册的设施的每模块注册，因此如果初始化在某个点失败，模块必须能自己退回所有东西. 如果你无法注销你获取的东西，内核就被置于一个不稳定状态；它包含了不存在的代码的内部指针. 这种情况下，经常地，唯一的方法就是重启系统. 在初始化错误发生时，你确实要小心地将事情做正确.

错误恢复有时用 goto 语句处理是最好的. 我们通常不愿使用 goto，但是在我们的观念里，这是一个它有用的地方. 在错误情形下小心使用 goto 可以去掉大量的复杂，过度对齐的，"结构形" 的逻辑. 因此，在内核里，goto 是处理错误经常用到，如这里显示的.

下面例子代码(使用设施注册和注销函数)在初始化在任何点失败时做得正确：

```
int __init my_init_function(void)
{
        int err;
        err = register_this(ptr1, "skull"); /* registration takes a pointer and a name */
        if (err)
                goto fail_this;
         err = register_that(ptr2, "skull");
        if (err)
                goto fail_that;
         err = register_those(ptr3, "skull");
        if (err)
                goto fail_those;
        return 0;         /* success */
fail_those:
        unregister_that(ptr2, "skull");
fail_that:
        unregister_this(ptr1, "skull");
fail_this:
        return err;       /* propagate the error */
```

```
}
```

这段代码试图注册 3 个(虚构的)设施. goto 语句在失败情况下使用, 在事情变坏之前只对之前已经成功注册的设施进行注销.

另一个选项, 不需要繁多的 goto 语句, 是跟踪已经成功注册的, 并且在任何出错情况下调用你的模块的清理函数. 清理函数只回卷那些已经成功完成的步骤. 然而这种选择, 需要更多代码和更多 CPU 时间, 因此在快速途径下, 你仍然依赖于 goto 作为最好的错误恢复工具.

my_init_function 的返回值, err, 是一个错误码. 在 Linux 内核里, 错误码是负数, 属于定义于 <linux/errno.h> 的集合. 如果你需要产生你自己的错误码代替你从其他函数得到的返回值, 你应当包含 <linux/errno.h> 以便使用符号式的返回值, 例如 -ENODEV, -ENOMEM, 等等. 返回适当的错误码总是一个好做法, 因为用户程序能够把它们转变为有意义的字串, 使用 perror 或者类似的方法.

显然, 模块清理函数必须撤销任何由初始化函数进行的注册, 并且惯例(但常常不是要求的)是按照注册时相反的顺序注销设施.

```
void __exit my_cleanup_function(void)
{
 unregister_those(ptr3, "skull");
 unregister_that(ptr2, "skull");
 unregister_this(ptr1, "skull");
 return;
}
```

如果你的初始化和清理比处理几项复杂, goto 方法可能变得难于管理, 因为所有的清理代码必须在初始化函数里重复, 包括几个混合的标号. 有时, 因此, 一种不同的代码排布证明更成功.

使代码重复最小和所有东西流线化, 你应当做的是无论何时发生错误都从初始化里调用清理函数. 清理函数接着必须在撤销它的注册前检查每一项的状态. 以最简单的形式, 代码看起来象这样:

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
        if (item1)
                release_thing(item1);
        if (item2)
                release_thing2(item2);
        if (stuff_ok)
                unregister_stuff();
        return;
}
```

```
int __init my_init(void)
{
        int err = -ENOMEM;

        item1 = allocate_thing(arguments);
        item2 = allocate_thing2(arguments2);
        if (!item2 || !item2)
                goto fail;

        err = register_stuff(item1, item2);
        if (!err)
                stuff_ok = 1;
        else
                goto fail;
        return 0; /* success */

fail:
        my_cleanup();
        return err;
}
```

如这段代码所示，你也许需要，也许不要外部的标志来标识初始化步骤的成功，要依赖你调用的注册/分配函数的语义．不管要不要标志，这种初始化会变得包含大量的项，常常比之前展示的技术要好．注意，但是，清理函数当由非退出代码调用时不能标志为 __exit，如同前面的例子．

2.7.3. 模块加载竞争

到目前，我们的讨论已来到一个模块加载的重要方面：竞争情况．如果你在如何编写你的初始化函数上不小心，你可能造成威胁到整个系统的稳定的情形．我们将在本书稍后讨论竞争情况；现在，快速提几点就足够了：

首先时你应该一直记住，内核的某些别的部分会在注册完成之后马上使用任何你注册的设施．这是完全可能的，换句话说，内核将调用进你的模块，在你的初始化函数仍然在运行时．所以你的代码必须准备好被调用，一旦它完成了它的第一个注册．不要注册任何设施，直到所有的需要支持那个设施的你的内部初始化已经完成．

你也必须考虑到如果你的初始化函数决定失败会发生什么，但是内核的一部分已经在使用你的模块已注册的设施．如果这种情况对你的模块是可能的，你应当认真考虑根本不要使初始化失败．毕竟，模块已清楚地成功输出一些有用的东西．如果初始化必须失败，必须小心地处理任何可能的在内核别处发生的操作，直到这些操作已完成．

2.8. 模块参数

驱动需要知道的几个参数因不同的系统而不同．从使用的设备号(如我们在下一章见到的)到驱动应当任何操作的几个方面．例如，SCSI 适配器的驱动常常有选项控制标记命令队列的使用，IDE 驱动允许用户控制 DMA 操作．如果你的驱动控制老的硬件，还需要被明确告知哪里去找硬件的 I/O 端口或者 I/O 内存地址．内核通过在加载驱动的模块时指定可变参数的值，支持这些要求．

这些参数的值可由 insmod 或者 modprobe 在加载时指定；后者也可以从它的配置文件(/etc/modprobe.conf)读取参数的值. 这些命令在命令行里接受几类规格的值. 作为演示这种能力的一种方法，想象一个特别需要的对本章开始的"hello world"模块(称为 hellop)的改进. 我们增加 2 个参数：一个整型值，称为 howmany，一个字符串称为 whom. 我们的特别多功能的模块就在加载时，欢迎 whom 不止一次，而是 howmany 次. 这样一个模块可以用这样的命令行加载：

```
insmod hellop howmany=10 whom="Mom"
```

一旦以那样的方式加载，hellop 会说 "hello，Mom" 10 次.

但是，在 insmod 可以修改模块参数前，模块必须使它们可用. 参数用 moudle_param 宏定义来声明，它定义在 moduleparam.h. module_param 使用了 3 个参数：变量名，它的类型，以及一个权限掩码用来做一个辅助的 sysfs 入口. 这个宏定义应当放在任何函数之外，典型地是出现在源文件的前面. 因此 hellop 将声明它的参数，并如下使得对 insmod 可用：

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

模块参数支持许多类型：

bool
invbool

　　一个布尔型(true 或者 false)值(相关的变量应当是 int 类型). invbool 类型颠倒了值，所以真值变成 false，反之亦然.

charp

　　一个字符指针值. 内存为用户提供的字串分配，指针因此设置.

int
long
short
uint
ulong
ushort

　　基本的变长整型值. 以 u 开头的是无符号值.

数组参数，用逗号间隔的列表提供的值，模块加载者也支持. 声明一个数组参数，使用：

```
module_param_array(name,type,num,perm);
```

这里 name 是你的数组的名子(也是参数名)， type 是数组元素的类型， num 是一个整型变量， perm 是通常的权限值. 如果数组参数在加载时设置， num 被设置成提供的数的个数. 模块加载者拒绝比数组能放下的多的值.

如果你确实需要一个没有出现在上面列表中的类型， 在模块代码里有钩子会允许你来定义它们; 任何使用它们的细节见 moduleparam.h. 所有的模块参数应当给定一个缺省值; insmod 只在用户明确告知它的时候才改变这些值. 模块可检查明显的参数， 通过对应它们的缺省值检查这些参数.

最后的 module_param 字段是一个权限值; 你应当使用 <linux/stat.h> 中定义的值. 这个值控制谁可以存取这些模块参数在 sysfs 中的表示. 如果 perm 被设为 0， 就根本没有 sysfs 项. 否则， 它出现在 /sys/module[5] 下面， 带有给定的权限. 使用 S_IRUGO 作为参数可以被所有人读取， 但是不能改变; S_IRUGO|S_IWUSR 允许 root 来改变参数. 注意， 如果一个参数被 sysfs 修改， 你的模块看到的参数值也改变了， 但是你的模块没有任何其他的通知. 你应当不要使模块参数可写， 除非你准备好检测这个改变并且因而作出反应.

2.9. 在用户空间做

一个第一次涉及内核问题的 Unix 程序员， 可能会紧张写一个模块. 编写一个用户程序来直接读写设备端口可能容易些.

确实， 有几个论据倾向于用户空间编程， 有时编写一个所谓的用户空间设备驱动对比钻研内核是一个明智的选择. 在本节， 我们讨论几个理由， 为什么你可能在用户空间编写驱动. 本书是关于内核空间驱动的， 但是， 所以我们不超越这个介绍性的讨论.

用户空间驱动的好处在于:

- 完整的 C 库可以连接. 驱动可以进行许多奇怪的任务， 不用依靠外面的程序(实现使用策略的工具程序， 常常随着驱动自身发布).
- 程序员可以在驱动代码上运行常用的调试器， 而不必走调试一个运行中的内核的弯路.
- 如果一个用户空间驱动挂起了， 你可简单地杀掉它. 驱动的问题不可能挂起整个系统， 除非被控制的硬件真的疯掉了.
- 用户内存是可交换的， 不象内核内存. 一个不常使用的却有很大一个驱动的设备不会占据别的程序可以用到的 RAM， 除了在它实际在用时.
- 一个精心设计的驱动程序仍然可以， 如同内核空间驱动， 允许对设备的并行存取.
- 如果你必须编写一个封闭源码的驱动， 用户空间的选项使你容易避免不明朗的许可的情况和改变的内核接口带来的问题.

例如， USB 驱动能够在用户空间编写; 看(仍然年幼) libusb 项目， 在 libusb.sourceforge.net 和 "gadgetfs" 在内核源码里. 另一个例子是 X 服务器: 它确切地知道它能处理哪些硬件， 哪些不能， 并且它提供图形资源给所有的 X 客户. 注意， 然而， 有一个缓慢但是固定的漂移向着基于 frame-buffer 的图形环境， X 服务器只是作为一个服务器， 基于一个内核空间的真实的设备驱动， 这个驱动负责真正的图形操作.

---

[5] 然而， 在本书写作时， 有讨论将参数移出 sysfs.

常常，用户空间驱动的编写者完成一个服务器进程，从内核接管作为单个代理的负责硬件控制的任务. 客户应用程序就可以连接到服务器来进行实际的操作; 因此，一个聪明的驱动经常可以允许对设备的并行存取. 这就是 X 服务器如何工作的.

但是用户空间的设备驱动的方法有几个缺点. 最重要的是:

- 中断在用户空间无法用. 在某些平台上有对这个限制的解决方法，例如在 IA32 体系上的 vm86 系统调用.
- 只可能通过内存映射 /dev/mem 来使用 DMA，而且只有特权用户可以这样做.
- 存取 I/O 端口只能在调用 ioperm 或者 iopl 之后. 此外，不是所有的平台支持这些系统调用，而存取/dev/port 可能太慢而无效率. 这些系统调用和设备文件都要求特权用户.
- 响应时间慢，因为需要上下文切换在客户和硬件之间传递信息或动作.
- 更不好的是，如果驱动已被交换到硬盘，响应时间会长到不可接受. 使用 mlock 系统调用可能会有帮助，但是常常的你将需要锁住许多内存页，因为一个用户空间程序依赖大量的库代码. mlock，也，限制在授权用户上.
- 最重要的设备不能在用户空间处理，包括但不限于，网络接口和块设备.

如你所见，用户空间驱动不能做的事情毕竟太多. 感兴趣的应用程序还是存在: 例如，对 SCSI 扫描器设备的支持(由 SANE 包实现)和 CD 刻录器 (由 cdrecord 和别的工具实现). 在两种情况下，用户级别的设备情况依赖 "SCSI generic" 内核驱动，它输出了低层的 SCSI 功能给用户程序，因此它们可以驱动它们自己的硬件.

一种在用户空间工作的情况可能是有意义的，当你开始处理新的没有用过的硬件时. 这样你可以学习去管理你的硬件，不必担心挂起整个系统. 一旦你完成了，在一个内核模块中封装软件就会是一个简单操作了.

2.10. 快速参考

本节总结了我们在本章接触到的内核函数，变量，宏定义，和 /proc 文件. 它的用意是作为一个参考. 每一项列都在相关头文件的后面，如果有. 从这里开始，在几乎每章的结尾会有类似一节，总结一章中介绍的新符号. 本节中的项通常以在本章中出现的顺序排列:

```
insmod
modprobe
rmmod
```

用户空间工具，加载模块到运行中的内核以及去除它们.

```
#include <linux/init.h>
module_init(init_function);
module_exit(cleanup_function);
```

指定模块的初始化和清理函数的宏定义.

```
__init
__initdata
__exit
```

__exitdata

函数(__init 和 __exit)和数据 (__initdata 和 __exitdata)的标记, 只用在模块初始化或者清理时间. 为初始化所标识的项可能会在初始化完成后丢弃; 退出的项可能被丢弃如果内核没有配置模块卸载. 这些标记通过使相关的目标在可执行文件的特定的 ELF 节里被替换来工作.

#include <linux/sched.h>

最重要的头文件中的一个. 这个文件包含很多驱动使用的内核 API 的定义, 包括睡眠函数和许多变量声明.

struct task_struct *current;

当前进程.

current->pid
current->comm

进程 ID 和 当前进程的命令名.

obj-m

一个 makefile 符号, 内核建立系统用来决定当前目录下的哪个模块应当被建立.

/sys/module
/proc/modules

/sys/module 是一个 sysfs 目录层次, 包含当前加载模块的信息. /proc/moudles 是旧式的, 那种信息的单个文件版本. 其中的条目包含了模块名, 每个模块占用的内存数量, 以及使用计数. 另外的字串追加到每行的末尾来指定标志, 对这个模块当前是活动的.

vermagic.o

来自内核源码目录的目标文件, 描述一个模块为之建立的环境.

#include <linux/module.h>

必需的头文件. 它必须在一个模块源码中包含.

#include <linux/version.h>

头文件, 包含在建立的内核版本信息.

LINUX_VERSION_CODE

整型宏定义, 对 #ifdef 版本依赖有用.

```
EXPORT_SYMBOL (symbol);
EXPORT_SYMBOL_GPL (symbol);
```

宏定义, 用来输出一个符号给内核. 第 2 种形式输出没有版本信息, 第 3 种限制输出给 GPL 许可的模块.

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

放置文档在目标文件的模块中.

```
module_init(init_function);
module_exit(exit_function);
```

宏定义, 声明一个模块的初始化和清理函数.

```
#include <linux/moduleparam.h>
module_param(variable, type, perm);
```

宏定义, 创建模块参数, 可以被用户在模块加载时调整(或者在启动时间, 对于内嵌代码). 类型可以是 bool, charp, int, invbool, short, ushort, uint, ulong, 或者 intarray.

```
#include <linux/kernel.h>
int printk(const char * fmt, ...);
```

内核代码的 printf 类似物.

第 3 章 字符驱动

本章的目的是编写一个完整的字符设备驱动. 我们开发一个字符驱动是因为这一类适合大部分简单硬件设备. 字符驱动也比块驱动易于理解(我们在后续章节接触). 我们的最终目的是编写一个模块化的字符驱动, 但是我们不会在本章讨论模块化的事情.

贯串本章, 我们展示从一个真实设备驱动提取的代码片段: scull(Simple Character Utility for Loading Localities). scull 是一个字符驱动, 操作一块内存区域好像它是一个设备. 在本章, 因为 scull 的这个怪特性, 我们可互换地使用设备这个词和"scull 使用的内存区".

scull 的优势在于它不依赖硬件. scull 只是操作一些从内核分配的内存. 任何人都可以编译和运行 scull, 并且 scull 在 Linux 运行的体系结构中可移植. 另一方面, 这个设备除了演示内核和字符驱动的接口和允许用户运行一些测试之外, 不做任何有用的事情.

3.1. scull 的设计

编写驱动的第一步是定义驱动将要提供给用户程序的能力(机制).因为我们的"设备"是计算机内存的一部分, 我们可自由做我们想做的事情. 它可以是一个顺序的或者随机存取的设备, 一个或多个设备, 等等.

为使 scull 作为一个模板来编写真实设备的真实驱动, 我们将展示给你如何在计算机内存上实现几个设备抽象, 每个有不同的个性.

scull 源码实现下面的设备. 模块实现的每种设备都被引用做一种类型.

scull0 到 scull3

> 4 个设备, 每个由一个全局永久的内存区组成. 全局意味着如果设备被多次打开, 设备中含有的数据由所有打开它的文件描述符共享. 永久意味着如果设备关闭又重新打开, 数据不会丢失. 这个设备用起来有意思, 因为它可以用惯常的命令来存取和测试, 例如 cp, cat, 以及 I/O 重定向.

scullpipe0 到 scullpipe3

> 4 个 FIFO (先入先出) 设备, 行为象管道. 一个进程读的内容来自另一个进程所写的. 如果多个进程读同一个设备, 它们竞争数据. scullpipe 的内部将展示阻塞读写和非阻塞读写如何实现, 而不必采取中断. 尽管真实的驱动使用硬件中断来同步它们的设备, 阻塞和非阻塞操作的主题是重要的并且与中断处理是分开的.(在第 10 章涉及).

scullsingle
scullpriv
sculluid
scullwuid

> 这些设备与 scull0 相似, 但是在什么时候允许打开上有一些限制. 第一个 (snullsingle) 只允许一次一个进程使用驱动, 而 scullpriv 对每个虚拟终端(或

者 X 终端会话)是私有的，因为每个控制台/终端上的进程有不同的内存区. sculluid 和 scullwuid 可以多次打开，但是一次只能是一个用户；前者返回一个"设备忙"错误，如果另一个用户锁着设备，而后者实现阻塞打开. 这些 scull 的变体可能看来混淆了策略和机制，但是它们值得看看，因为一些实际设备需要这类管理.

每个 scull 设备演示了驱动的不同特色，并且呈现了不同的难度. 本章涉及 scull0 到 scull3 的内部；更高级的设备在第 6 章涉及. scullpipe 在"一个阻塞 I/O 例子"一节中描述，其他的在"设备文件上的存取控制"中描述.

3.2. 主次编号

字符设备通过文件系统中的名子来存取. 那些名子称为文件系统的特殊文件，或者设备文件，或者文件系统的简单结点；惯例上它们位于 /dev 目录. 字符驱动的特殊文件由使用 ls -l 的输出的第一列的"c"标识. 块设备也出现在 /dev 中，但是它们由"b"标识. 本章集中在字符设备，但是下面的很多信息也适用于块设备.

如果你发出 ls -l 命令，你会看到在设备文件项中有 2 个数(由一个逗号分隔)在最后修改日期前面，这里通常是文件长度出现的地方. 这些数字是给特殊设备的主次设备编号. 下面的列表显示了一个典型系统上出现的几个设备. 它们的主编号是 1，4，7，和 10，而次编号是 1，3，5，64，65，和 129.

```
crw-rw-rw- 1 root  root   1,   3 Apr 11  2002 null
crw------- 1 root  root  10,   1 Apr 11  2002 psaux
crw------- 1 root  root   4,   1 Oct 28 03:04 tty1
crw-rw-rw- 1 root  tty    4,  64 Apr 11  2002 ttys0
crw-rw---- 1 root  uucp   4,  65 Apr 11  2002 ttyS1
crw--w---- 1 vcsa  tty    7,   1 Apr 11  2002 vcs1
crw--w---- 1 vcsa  tty    7,129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root  root   1,   5 Apr 11  2002 zero
```

传统上，主编号标识设备相连的驱动. 例如，/dev/null 和 /dev/zero 都由驱动 1 来管理，而虚拟控制台和串口终端都由驱动 4 管理；同样，vcs1 和 vcsa1 设备都由驱动 7 管理. 现代 Linux 内核允许多个驱动共享主编号，但是你看到的大部分设备仍然按照一个主编号一个驱动的原则来组织.

次编号被内核用来决定引用哪个设备. 依据你的驱动是如何编写的(如同我们下面见到的)，你可以从内核得到一个你的设备的直接指针，或者可以自己使用次编号作为本地设备数组的索引. 不论哪个方法，内核自己几乎不知道次编号的任何事情，除了它们指向你的驱动实现的设备.

3.2.1. 设备编号的内部表示

在内核中，dev_t 类型(在 <linux/types.h>中定义)用来持有设备编号 -- 主次部分都包括. 对于 2.6.0 内核，dev_t 是 32 位的量，12 位用作主编号，20 位用作次编号. 你的代码应当，当然，对于设备编号的内部组织从不做任何假设；相反，应当利用在 <linux/kdev_t.h>中的一套宏定义. 为获得一个 dev_t 的主或者次编号，使用:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

相反，如果你有主次编号，需要将其转换为一个 dev_t，使用：

```
MKDEV(int major, int minor);
```

注意，2.6 内核能容纳有大量设备，而以前的内核版本限制在 255 个主编号和 255 个次编号。有人认为这么宽的范围在很长时间内是足够的，但是计算领域被这个特性的错误假设搞乱了。因此你应当希望 dev_t 的格式将来可能再次改变；但是，如果你仔细编写你的驱动，这些变化不会是一个问题。

3.2.2. 分配和释放设备编号

在建立一个字符驱动时你的驱动需要做的第一件事是获取一个或多个设备编号来使用。为此目的的必要的函数是 register_chrdev_region，在 <linux/fs.h>中声明：

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

这里，first 是你要分配的起始设备编号。first 的次编号部分常常是 0，但是没有要求是那个效果。count 是你请求的连续设备编号的总数。注意，如果 count 太大，你要求的范围可能溢出到下一个次编号；但是只要你要求的编号范围可用，一切都仍然会正确工作。最后，name 是应当连接到这个编号范围的设备的名子；它会出现在 /proc/devices 和 sysfs 中。

如同大部分内核函数，如果分配成功进行，register_chrdev_region 的返回值是 0。出错的情况下，返回一个负的错误码，你不能存取请求的区域。

如果你确实事先知道你需要哪个设备编号，register_chrdev_region 工作得好。然而，你常常不会知道你的设备使用哪个主编号；在 Linux 内核开发社团中一直努力使用动态分配设备编号。内核会乐于动态为你分配一个主编号，但是你必须使用一个不同的函数来请求这个分配。

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count,
char *name);
```

使用这个函数，dev 是一个只输出的参数，它在函数成功完成时持有你的分配范围的第一个数。fisetminor 应当是请求的第一个要用的次编号；它常常是 0。count 和 name 参数如同给 request_chrdev_region 的一样。

不管你任何分配你的设备编号，你应当在不再使用它们时释放它。设备编号的释放使用：

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

调用 unregister_chrdev_region 的地方常常是你的模块的 cleanup 函数。

上面的函数分配设备编号给你的驱动使用，但是它们不告诉内核你实际上会对这些编号做什么。在用户空间程序能够存取这些设备号中一个之前，你的驱动需要连接它们到它的实

现设备操作的内部函数上. 我们将描述如何简短完成这个连接, 但首先顾及一些必要的枝节问题.

3.2.3. 主编号的动态分配

一些主设备编号是静态分派给最普通的设备的. 一个这些设备的列表在内核源码树的 Documentation/devices.txt 中. 分配给你的新驱动使用一个已经分配的静态编号的机会很小, 但是, 并且新编号没在分配. 因此, 作为一个驱动编写者, 你有一个选择: 你可以简单地捡一个看来没有用的编号, 或者你以动态方式分配主编号. 只要你是你的驱动的唯一用户就可以捡一个编号用; 一旦你的驱动更广泛的被使用了, 一个随机捡来的主编号将导致冲突和麻烦.

因此, 对于新驱动, 我们强烈建议你使用动态分配来获取你的主设备编号, 而不是随机选取一个当前空闲的编号. 换句话说, 你的驱动应当几乎肯定地使用 alloc_chrdev_region, 不是 register_chrdev_region.

动态分配的缺点是你无法提前创建设备节点, 因为分配给你的模块的主编号会变化. 对于驱动的正常使用, 这不是问题, 因为一旦编号分配了, 你可从 /proc/devices 中读取它.[6][6]

为使用动态主编号来加载一个驱动, 因此, 可使用一个简单的脚本来代替调用 insmod, 在调用 insmod 后, 读取 /proc/devices 来创建特殊文件.

一个典型的 /proc/devices 文件看来如下:

```
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 ttyS
 6 lp
 7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 180 usb

Block devices:
 2 fd
 8 sd
 11 sr
 65 sd
 66 sd
```

[6] [6] 从 sysfs 中能获取更好的设备信息, 在基于 2.6 的系统通常加载于 /sys. 但是使 scull 通过 sysfs 输出信息超出了本章的范围; 我们在 14 章中回到这个主题.

因此加载一个已经安排了一个动态编号的模块的脚本，可以使用一个工具来编写，如 awk，来从 /proc/devices 获取信息以创建 /dev 中的文件.

下面的脚本，snull_load，是 scull 发布的一部分. 以模块发布的驱动的用户可以从系统的 rc.local 文件中调用这样一个脚本，或者在需要模块时手工调用它.

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./$module.ko $* || exit 1

# remove stale nodes
rm -f /dev/${device}[0-3]

major=$(awk "\\$2==\"$module\" {print \\$1}" /proc/devices)
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

这个脚本可以通过重定义变量和调整 mknod 行来适用于另外的驱动. 这个脚本仅仅展示了创建 4 个设备，因为 4 是 scull 源码中缺省的.

脚本的最后几行可能有些模糊:为什么改变设备的组和模式? 理由是这个脚本必须由超级用户运行，因此新建的特殊文件由 root 拥有. 许可位缺省的是只有 root 有写权限，而任何人可以读. 通常，一个设备节点需要一个不同的存取策略，因此在某些方面别人的存取权限必须改变. 我们的脚本缺省是给一个用户组存取，但是你的需求可能不同. 在第 6 章的"设备文件的存取控制"一节中，sculluid 的代码演示了驱动如何能够强制它自己的对设备存取的授权.

还有一个 scull_unload 脚本来清理 /dev 目录并去除模块.

作为对使用一对脚本来加载和卸载的另外选择，你可以编写一个 init 脚本，准备好放在你的发布使用这些脚本的目录中. [7][7]作为 scull 源码的一部分，我们提供了一个相当完整

[7] [7] Linux Standard Base 指出 init 脚本应当放在 /etc/init.d，但是一些发布仍然放在别处. 另外，如果你的脚本在启动时运行，你需要从合适的运行级别目录做一个连接给它(也就是，.../rc3.d).

和可配置的 init 脚本例子，称为 scull.init；它接受传统的参数 -- start，stop，和 restart -- 并且完成 scull_load 和 scull_unload 的角色.

如果反复创建和销毁 /dev 节点，听来过分了，有一个有用的办法. 如果你在加载和卸载单个驱动，你可以在你第一次使用你的脚本创建特殊文件之后，只使用 rmmod 和 insmod: 这样动态编号不是随机的. [8]8并且你每次都可以使用所选的同一个编号，如果你不加载任何别的动态模块. 在开发中避免长脚本是有用的. 但是这个技巧，显然不能扩展到一次多于一个驱动.

安排主编号最好的方式，我们认为，是缺省使用动态分配，而留给自己在加载时指定主编号的选项权，或者甚至在编译时. scull 实现以这种方式工作；它使用一个全局变量，scull_major，来持有选定的编号(还有一个 scull_minor 给次编号). 这个变量初始化为 SCULL_MAJOR，定义在 scull.h. 发布的源码中的 SCULL_MAJOR 的缺省值是 0，意思是"使用动态分配". 用户可以接受缺省值或者选择一个特殊主编号，或者在编译前修改宏定义或者在 insmod 命令行指定一个值给 scull_major. 最后，通过使用 scull_load 脚本，用户可以在 scull_load 的命令行传递参数给 insmod. [9]9

这是我们用在 scull 的源码中获取主编号的代码:

```
if (scull_major) {
 dev = MKDEV(scull_major, scull_minor);
 result = register_chrdev_region(dev, scull_nr_devs, "scull");
} else {
 result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
 scull_major = MAJOR(dev);
}
if (result < 0) {
 printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
 return result;
}
```

本书使用的几乎所有例子驱动使用类似的代码来分配它们的主编号.

3.3. 一些重要数据结构

如同你想象的，注册设备编号仅仅是驱动代码必须进行的诸多任务中的第一个. 我们将很快看到其他重要的驱动组件，但首先需要涉及一个别的. 大部分的基础性的驱动操作包括 3 个重要的内核数据结构，称为 file_operations，file，和 inode. 需要对这些结构的基本了解才能够做大量感兴趣的事情，因此我们现在在进入如何实现基础性驱动操作的细节之前，会快速查看每一个.

3.3.1. 文件操作

8 [8] 尽管某些内核开发者已警告说将来就会这样做.

9 [9] init 脚本 scull.init 不在命令行中接受驱动选项，但是它支持一个配置文件，因为它被设计来在启动和关机时自动使用.

到现在，我们已经保留了一些设备编号给我们使用，但是我们还没有连接任何我们设备操作到这些编号上. file_operation 结构是一个字符驱动如何建立这个连接. 这个结构，定义在 <linux/fs.h>，是一个函数指针的集合. 每个打开文件(内部用一个 file 结构来代表，稍后我们会查看)与它自身的函数集合相关连(通过包含一个称为 f_op 的成员，它指向一个 file_operations 结构). 这些操作大部分负责实现系统调用，因此，命名为 open，read，等等. 我们可以认为文件是一个"对象"并且其上的函数操作称为它的"方法"，使用面向对象编程的术语来表示一个对象声明的用来操作对象的动作. 这是我们在 Linux 内核中看到的第一个面向对象编程的现象，后续章中我们会看到更多.

传统上，一个 file_operation 结构或者其一个指针称为 fops(或者它的一些变体). 结构中的每个成员必须指向驱动中的函数，这些函数实现一个特别的操作，或者对于不支持的操作留置为 NULL. 当指定为 NULL 指针时内核的确切的行为是每个函数不同的，如同本节后面的列表所示.

下面的列表介绍了一个应用程序能够在设备上调用的所有操作. 我们已经试图保持列表简短，这样它可作为一个参考，只是总结每个操作和在 NULL 指针使用时的缺省内核行为.

在你通读 file_operations 方法的列表时，你会注意到不少参数包含字串 __user. 这种注解是一种文档形式，注意，一个指针是一个不能被直接解引用的用户空间地址. 对于正常的编译，__user 没有效果，但是它可被外部检查软件使用来找出对用户空间地址的错误使用.

本章剩下的部分，在描述一些其他重要数据结构后，解释了最重要操作的角色并且给了提示，告诫和真实代码例子. 我们推迟讨论更复杂的操作到后面章节，因为我们还不准备深入如内存管理，阻塞操作，和异步通知.

struct module *owner

> 第一个 file_operations 成员根本不是一个操作；它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中，它被简单初始化为 THIS_MODULE，一个在 <linux/module.h> 中定义的宏.

loff_t (*llseek) (struct file *, loff_t, int);

> llseek 方法用作改变文件中的当前读/写位置，并且新位置作为(正的)返回值. loff_t 参数是一个"long offset"，并且就算在 32 位平台上也至少 64 位宽. 错误由一个负返回值指示. 如果这个函数指针是 NULL，seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器(在"file 结构" 一节中描述).

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);

> 用来从设备中获取数据. 在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid argument") 失败. 一个非负返回值代表了成功读取的字节数(返回值是一个 "signed size" 类型，常常是目标平台本地的整数类型).

ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);

初始化一个异步读 -- 可能在函数返回前不结束的读操作. 如果这个方法是 NULL, 所有的操作会由 read 代替进行(同步地).

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

发送数据给设备. 如果 NULL, -EINVAL 返回给调用 write 系统调用的程序. 如果非负, 返回值代表成功写的字节数.

ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);

初始化设备上的一个异步写.

int (*readdir) (struct file *, void *, filldir_t);

对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对文件系统有用.

unsigned int (*poll) (struct file *, struct poll_table_struct *);

poll 方法是 3 个系统调用的后端: poll, epoll, 和 select, 都用作查询对一个或多个文件描述符的读或写是否会阻塞. poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的, 并且, 可能地, 提供给内核信息用来使调用进程睡眠直到 I/O 变为可能. 如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道, 这不是读也不是写). 另外, 几个 ioctl 命令被内核识别而不必引用 fops 表. 如果设备不提供 ioctl 方法, 对于任何未事先定义的请求(-ENOTTY, "设备无这样的 ioctl"), 系统调用返回一个错误.

int (*mmap) (struct file *, struct vm_area_struct *);

mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL, mmap 系统调用返回 -ENODEV.

int (*open) (struct inode *, struct file *);

尽管这常常是对设备文件进行的第一个操作, 不要求驱动声明一个对应的方法. 如果这个项是 NULL, 设备打开一直成功, 但是你的驱动不会得到通知.

int (*flush) (struct file *);

flush 操作在进程关闭它的设备文件描述符的拷贝时调用; 它应当执行(并且等待)设备的任何未完成的操作. 这个必须不要和用户查询请求的 fsync 操作混淆了. 当前, flush 在很少驱动中使用; SCSI 磁带驱动使用它, 例如, 为确保所有写的数据在设备关闭前写到磁带上. 如果 flush 为 NULL, 内核简单地忽略用户应用程序的请求.

int (*release) (struct inode *, struct file *);

在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.

int (*fsync) (struct file *, struct dentry *, int);

这个方法是 fsync 系统调用的后端, 用户调用来刷新任何挂着的数据. 如果这个指针是 NULL, 系统调用返回 -EINVAL.

int (*aio_fsync)(struct kiocb *, int);

这是 fsync 方法的异步版本.

int (*fasync) (int, struct file *, int);

这个操作用来通知设备它的 FASYNC 标志的改变. 异步通知是一个高级的主题, 在第 6 章中描述. 这个成员可以是NULL 如果驱动不支持异步通知.

int (*lock) (struct file *, int, struct file_lock *);

lock 方法用来实现文件加锁; 加锁对常规文件是必不可少的特性, 但是设备驱动几乎从不实现它.

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);

这些方法实现发散/汇聚读和写操作. 应用程序偶尔需要做一个包含多个内存区的单个读或写操作; 这些系统调用允许它们这样做而不必对数据进行额外拷贝. 如果这些函数指针为 NULL, read 和 write 方法被调用(可能多于一次).

ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);

这个方法实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个. 例如, 它被一个需要发送文件内容到一个网络连接的 web 服务器使用. 设备驱动常常使 sendfile 为 NULL.

ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);

sendpage 是 sendfile 的另一半; 它由内核调用来发送数据, 一次一页, 到对应的文件. 设备驱动实际上不实现 sendpage.

unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);

这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中. 这个任务通常由内存管理代码进行; 这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求. 大部分驱动可以置这个方法为 NULL.[10] 10

int (*check_flags)(int)

这个方法允许模块检查传递给 fnctl(F_SETFL...) 调用的标志.

int (*dir_notify)(struct file *, unsigned long);

这个方法在应用程序使用 fcntl 来请求目录改变通知时调用. 只对文件系统有用; 驱动不需要实现 dir_notify.

scull 设备驱动只实现最重要的设备方法. 它的 file_operations 结构是如下初始化的:

```
struct file_operations scull_fops = {
 .owner =  THIS_MODULE,
 .llseek =  scull_llseek,
 .read =  scull_read,
 .write =  scull_write,
 .ioctl =  scull_ioctl,
 .open =  scull_open,
 .release =  scull_release,
};
```

这个声明使用标准的 C 标记式结构初始化语法. 这个语法是首选的, 因为它使驱动在结构定义的改变之间更加可移植, 并且, 有争议地, 使代码更加紧凑和可读. 标记式初始化允许结构成员重新排序; 在某种情况下, 真实的性能提高已经实现, 通过安放经常使用的成员的指针在相同硬件高速存储行中.

3.3.2. 文件结构

struct file, 定义于 <linux/fs.h>, 是设备驱动中第二个最重要的数据结构. 注意 file 与用户空间程序的 FILE 指针没有任何关系. 一个 FILE 定义在 C 库中, 从不出现在内核代码中. 一个 struct file, 另一方面, 是一个内核结构, 从不出现在用户程序中.

文件结构代表一个打开的文件. (它不特定给设备驱动; 系统中每个打开的文件有一个关联的 struct file 在内核空间). 它由内核在 open 时创建, 并传递给在文件上操作的任何函数, 直到最后的关闭. 在文件的所有实例都关闭后, 内核释放这个数据结构.

在内核源码中, struct file 的指针常常称为 file 或者 filp("file pointer"). 我们将一直称这个指针为 filp 以避免和结构自身混淆. 因此, file 指的是结构, 而 filp 是结构指针.

10 [10] 注意, release 不是每次进程调用 close 时都被调用. 无论何时共享一个文件结构(例如, 在一个 fork 或 dup 之后), release 不会调用直到所有的拷贝都关闭了. 如果你需要在任一拷贝关闭时刷新挂着的数据, 你应当实现 flush 方法.

struct file 的最重要成员在这展示. 如同在前一节, 第一次阅读可以跳过这个列表. 但是, 在本章后面, 当我们面对一些真实 C 代码时, 我们将更详细讨论这些成员.

mode_t f_mode;

文件模式确定文件是可读的或者是可写的(或者都是), 通过位 FMODE_READ 和 FMODE_WRITE. 你可能想在你的 open 或者 ioctl 函数中检查这个成员的读写许可, 但是你不需要检查读写许可, 因为内核在调用你的方法之前检查. 当文件还没有为那种存取而打开时读或写的企图被拒绝, 驱动甚至不知道这个情况.

loff_t f_pos;

当前读写位置. loff_t 在所有平台都是 64 位(在 gcc 术语里是 long long). 驱动可以读这个值, 如果它需要知道文件中的当前位置, 但是正常地不应该改变它; 读和写应当使用它们作为最后参数而收到的指针来更新一个位置, 代替直接作用于 filp->f_pos. 这个规则的一个例外是在 llseek 方法中, 它的目的就是改变文件位置.

unsigned int f_flags;

这些是文件标志, 例如 O_RDONLY, O_NONBLOCK, 和 O_SYNC. 驱动应当检查 O_NONBLOCK 标志来看是否是请求非阻塞操作(我们在第一章的"阻塞和非阻塞操作"一节中讨论非阻塞 I/O); 其他标志很少使用. 特别地, 应当检查读/写许可, 使用 f_mode 而不是 f_flags. 所有的标志在头文件 <linux/fcntl.h> 中定义.

struct file_operations *f_op;

和文件关联的操作. 内核安排指针作为它的 open 实现的一部分, 接着读取它当它需要分派任何的操作时. filp->f_op 中的值从不由内核保存为后面的引用; 这意味着你可改变你的文件关联的文件操作, 在你返回调用者之后新方法会起作用. 例如, 关联到主编号 1 (/dev/null, /dev/zero, 等等)的 open 代码根据打开的次编号来替代 filp->f_op 中的操作. 这个做法允许实现几种行为, 在同一个主编号下而不必在每个系统调用中引入开销. 替换文件操作的能力是面向对象编程的"方法重载"的内核对等体.

void *private_data;

open 系统调用设置这个指针为 NULL, 在为驱动调用 open 方法之前. 你可自由使用这个成员或者忽略它; 你可以使用这个成员来指向分配的数据, 但是接着你必须记住在内核销毁文件结构之前, 在 release 方法中释放那个内存. private_data 是一个有用的资源, 在系统调用间保留状态信息, 我们大部分例子模块都使用它.

struct dentry *f_dentry;

关联到文件的目录入口(dentry)结构. 设备驱动编写者正常地不需要关心 dentry 结构, 除了作为 filp->f_dentry->d_inode 存取 inode 结构.

真实结构有多几个成员, 但是它们对设备驱动没有用处. 我们可以安全地忽略这些成员, 因为驱动从不创建文件结构; 它们真实存取别处创建的结构.

3.3.3. inode 结构

inode 结构由内核在内部用来表示文件. 因此, 它和代表打开文件描述符的文件结构是不同的. 可能有代表单个文件的多个打开描述符的许多文件结构, 但是它们都指向一个单个 inode 结构.

inode 结构包含大量关于文件的信息. 作为一个通用的规则, 这个结构只有 2 个成员对于编写驱动代码有用:

dev_t i_rdev;

　　对于代表设备文件的节点, 这个成员包含实际的设备编号.

struct cdev *i_cdev;

　　struct cdev 是内核的内部结构, 代表字符设备; 这个成员包含一个指针, 指向这个结构, 当节点指的是一个字符设备文件时.

i_rdev 类型在 2.5 开发系列中改变了, 破坏了大量的驱动. 作为一个鼓励更可移植编程的方法, 内核开发者已经增加了 2 个宏, 可用来从一个 inode 中获取主次编号:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

为了不要被下一次改动抓住, 应当使用这些宏代替直接操作 i_rdev.

3.4. 字符设备注册

如我们提过的, 内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用你的设备操作前, 你编写分配并注册一个或几个这些结构. [11][11]为此, 你的代码应当包含 <linux/cdev.h>, 这个结构和它的关联帮助函数定义在这里.

有 2 种方法来分配和初始化一个这些结构. 如果你想在运行时获得一个独立的 cdev 结构, 你可以为此使用这样的代码:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

但是, 偶尔你会想将 cdev 结构嵌入一个你自己的设备特定的结构; scull 这样做了. 在这种情况下, 你应当初始化你已经分配的结构, 使用:

[11] [11] 有一个早些的机制以避免使用 cdev 结构(我们在"老方法"一节中讨论).但是, 新代码应当使用新技术.

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

任一方法，有一个其他的 struct cdev 成员你需要初始化. 象 file_operations 结构，struct cdev 有一个拥有者成员，应当设置为 THIS_MODULE. 一旦 cdev 结构建立，最后的步骤是把它告诉内核，调用：

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

这里，dev 是 cdev 结构，num 是这个设备响应的第一个设备号，count 是应当关联到设备的设备号的数目. 常常 count 是 1，但是有多个设备号对应于一个特定的设备的情形. 例如，设想 SCSI 磁带驱动，它允许用户空间来选择操作模式(例如密度)，通过安排多个次编号给每一个物理设备.

在使用 cdev_add 是有几个重要事情要记住. 第一个是这个调用可能失败. 如果它返回一个负的错误码，你的设备没有增加到系统中. 它几乎会一直成功，但是，并且带起了其他的点：cdev_add 一返回，你的设备就是"活的"并且内核可以调用它的操作. 除非你的驱动完全准备好处理设备上的操作，你不应当调用 cdev_add.

为从系统去除一个字符设备，调用：

```
void cdev_del(struct cdev *dev);
```

显然，你不应当在传递给 cdev_del 后存取 cdev 结构.

3.4.1. scull 中的设备注册

在内部，scull 使用一个 struct scull_dev 类型的结构表示每个设备. 这个结构定义为：

```
struct scull_dev {
 struct scull_qset *data;  /* Pointer to first quantum set */
 int quantum;  /* the current quantum size */
 int qset;  /* the current array size */
 unsigned long size;  /* amount of data stored here */
 unsigned int access_key;  /* used by sculluid and scullpriv */
 struct semaphore sem;  /* mutual exclusion semaphore  */
 struct cdev cdev; /* Char device structure */
};
```

我们在遇到它们时讨论结构中的各个成员，但是现在，我们关注于 cdev，我们的设备与内核接口的 struct cdev. 这个结构必须初始化并且如上所述添加到系统中；处理这个任务的 scull 代码是：

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
 int err, devno = MKDEV(scull_major, scull_minor + index);

 cdev_init(&dev->cdev, &scull_fops);
 dev->cdev.owner = THIS_MODULE;
 dev->cdev.ops = &scull_fops;
 err = cdev_add (&dev->cdev, devno, 1);
```

```
 /* Fail gracefully if need be */
 if (err)
 printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

因为 cdev 结构嵌在 struct scull_dev 里面, cdev_init 必须调用来进行那个结构的初始化.

3.4.2. 老方法

如果你深入浏览 2.6 内核的大量驱动代码, 你可能注意到有许多字符驱动不使用我们刚刚描述过的 cdev 接口. 你见到的是还没有更新到 2.6 内核接口的老代码. 因为那个代码实际上能用, 这个更新可能很长时间不会发生. 为完整, 我们描述老的字符设备注册接口, 但是新代码不应当使用它; 这个机制在将来内核中可能会消失.

注册一个字符设备的经典方法是使用:

```
int register_chrdev(unsigned int major, const char *name, struct
file_operations *fops);
```

这里, major 是感兴趣的主编号, name 是驱动的名子(出现在 /proc/devices), fops 是缺省的 file_operations 结构. 一个对 register_chrdev 的调用为给定的主编号注册 0 - 255 的次编号, 并且为每一个建立一个缺省的 cdev 结构. 使用这个接口的驱动必须准备好处理对所有 256 个次编号的 open 调用(不管它们是否对应真实设备), 它们不能使用大于 255 的主或次编号.

如果你使用 register_chrdev, 从系统中去除你的设备的正确的函数是:

```
int unregister_chrdev(unsigned int major, const char *name);
```

major 和 name 必须和传递给 register_chrdev 的相同, 否则调用会失败.

3.5. open 和 release

到此我们已经快速浏览了这些成员, 我们开始在真实的 scull 函数中使用它们.

3.5.1. open 方法

open 方法提供给驱动来做任何的初始化来准备后续的操作. 在大部分驱动中, open 应当进行下面的工作:

- 检查设备特定的错误(例如设备没准备好, 或者类似的硬件错误
- 如果它第一次打开, 初始化设备
- 如果需要, 更新 f_op 指针.
- 分配并填充要放进 filp->private_data 的任何数据结构

但是, 事情的第一步常常是确定打开哪个设备. 记住 open 方法的原型是:

```
int (*open)(struct inode *inode, struct file *filp);
```

inode 参数有我们需要的信息,以它的 i_cdev 成员的形式, 里面包含我们之前建立的 cdev 结构. 唯一的问题是通常我们不想要 cdev 结构本身, 我们需要的是包含 cdev 结构的 scull_dev 结构. C 语言使程序员玩弄各种技巧来做这种转换; 但是, 这种技巧编程是易出错的, 并且导致别人难于阅读和理解代码. 幸运的是, 在这种情况下, 内核 hacker 已经为我们实现了这个技巧, 以 container_of 宏的形式, 在 <linux/kernel.h> 中定义:

```
container_of(pointer, container_type, container_field);
```

这个宏使用一个指向 container_field 类型的成员的指针, 它在一个 container_type 类型的结构中, 并且返回一个指针指向包含结构. 在 scull_open, 这个宏用来找到适当的设备结构:

```
struct scull_dev *dev; /* device information */
dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* for other methods */
```

一旦它找到 scull_dev 结构, scull 在文件结构的 private_data 成员中存储一个它的指针, 为以后更易存取.

识别打开的设备的另外的方法是查看存储在 inode 结构的次编号. 如果你使用 register_chrdev 注册你的设备, 你必须使用这个技术. 确认使用 iminor 从 inode 结构中获取次编号, 并且确定它对应一个你的驱动真正准备好处理的设备.

scull_open 的代码(稍微简化过)是:

```
int scull_open(struct inode *inode, struct file *filp)
{
        struct scull_dev *dev; /* device information */
        dev = container_of(inode->i_cdev, struct scull_dev, cdev);
        filp->private_data = dev; /* for other methods */

        /* now trim to 0 the length of the device if open was write-only */
        if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
        {
                scull_trim(dev); /* ignore errors */
        }
        return 0; /* success */
}
```

代码看来相当稀疏, 因为在调用 open 时它没有做任何特别的设备处理. 它不需要, 因为 scull 设备设计为全局的和永久的. 特别地, 没有如"在第一次打开时初始化设备"等动作, 因为我们不为 scull 保持打开计数.

唯一在设备上的真实操作是当设备为写而打开时将它截取为长度为 0. 这样做是因为, 在设计上, 用一个短的文件覆盖一个 scull 设备导致一个短的设备数据区. 这类似于为写而打开一个常规文件, 将其截短为 0. 如果设备为读而打开, 这个操作什么都不做.

在我们查看其他 scull 特性的代码时将看到一个真实的初始化如何起作用的.

3.5.2. release 方法

release 方法的角色是 open 的反面. 有时你会发现方法的实现称为 device_close, 而不是 device_release. 任一方式, 设备方法应当进行下面的任务:

- 释放 open 分配在 filp->private_data 中的任何东西
- 在最后的 close 关闭设备

scull 的基本形式没有硬件去关闭, 因此需要的代码是最少的:[12] [12]

```
int scull_release(struct inode *inode, struct file *filp)
{
 return 0;
}
```

你可能想知道当一个设备文件关闭次数超过它被打开的次数会发生什么. 毕竟, dup 和 fork 系统调用不调用 open 来创建打开文件的拷贝; 每个拷贝接着在程序终止时被关闭. 例如, 大部分程序不打开它们的 stdin 文件(或设备), 但是它们都以关闭它结束. 当一个打开的设备文件已经真正被关闭时驱动如何知道?

答案简单: 不是每个 close 系统调用引起调用 release 方法. 只有真正释放设备数据结构的调用会调用这个方法 -- 因此得名. 内核维持一个文件结构被使用多少次的计数. fork 和 dup 都不创建新文件(只有 open 这样); 它们只递增正存在的结构中的计数. close 系统调用仅在文件结构计数掉到 0 时执行 release 方法, 这在结构被销毁时发生. release 方法和 close 系统调用之间的这种关系保证了你的驱动一次 open 只看到一次 release.

注意, flush 方法在每次应用程序调用 close 时都被调用. 但是, 很少驱动实现 flush, 因为常常在 close 时没有什么要做, 除非调用 release.

如你会想到的, 前面的讨论即便是应用程序没有明显地关闭它打开的文件也适用: 内核在进程 exit 时自动关闭了任何文件, 通过在内部使用 close 系统调用.

3.6. scull 的内存使用

在介绍读写操作前, 我们最好看看如何以及为什么 scull 进行内存分配. "如何"是需要全面理解代码, "为什么"演示了驱动编写者需要做的选择, 尽管 scull 明确地不是典型设备.

本节只处理 scull 中的内存分配策略, 不展示给你编写真正驱动需要的硬件管理技能. 这些技能在第 9 章和第 10 章介绍. 因此, 你可跳过本章, 如果你不感兴趣于理解面向内存的 scull 驱动的内部工作.

---

[12] [12] 其他风味的设备由不同的函数关闭, 因为 scull_open 为每个设备替换了不同的 filp->f_op. 我们在介绍每种风味时再讨论它们.

scull 使用的内存区，也称为一个设备，长度可变．你写的越多，它增长越多；通过使用一个短文件覆盖设备来进行修整．

scull 驱动引入 2 个核心函数来管理 Linux 内核中的内存．这些函数，定义在 <linux/slab.h>，是：

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

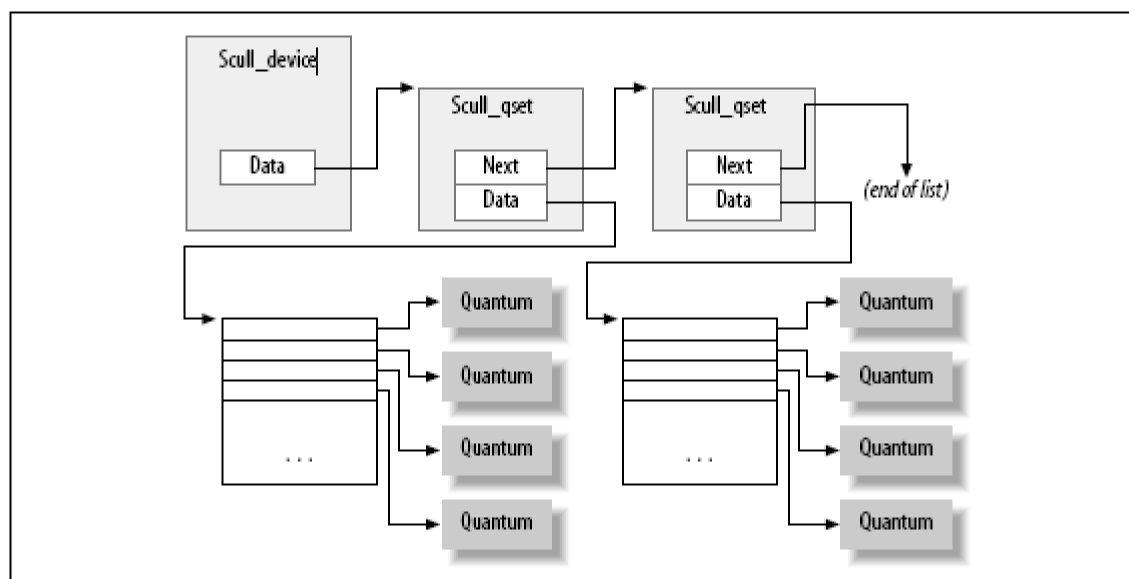
对 kmalloc 的调用试图分配 size 字节的内存；返回值是指向那个内存的指针或者如果分配失败为 NULL．flags 参数用来描述内存应当如何分配；我们在第 8 章详细查看这些标志．对于现在，我们一直使用 GFP_KERNEL．分配的内存应当用 kfree 来释放．你应当从不传递任何不是从 kmalloc 获得的东西给 kfree．但是，传递一个 NULL 指针给 kfree 是合法的．

kmalloc 不是最有效的分配大内存区的方法(见第 8 章)，所以挑选给 scull 的实现不是一个特别巧妙的．一个巧妙的源码实现可能更难阅读，而本节的目标是展示读和写，不是内存管理．这是为什么代码只是使用 kmalloc 和 kfree 而不依靠整页的分配，尽管这个方法会更有效．

在 flip 一边，我们不想限制"设备"区的大小，由于理论上的和实践上的理由．理论上，给在被管理的数据项施加武断的限制总是个坏想法．实践上，scull 可用来暂时地吃光你系统中的内存，以便运行在低内存条件下的测试．运行这样的测试可能会帮助你理解系统的内部．你可以使用命令 cp /dev/zero /dev/scull0 来用 scull 吃掉所有的真实 RAM，并且你可以使用 dd 工具来选择贝多少数据给 scull 设备．

在 scull，每个设备是一个指针链表，每个都指向一个 scull_dev 结构．每个这样的结构，缺省地，指向最多 4 兆字节，通过一个中间指针数组．发行代码使用一个 1000 个指针的数组指向每个 4000 字节的区域．我们称每个内存区域为一个量子，数组(或者它的长度)为一个量子集．一个 scull 设备和它的内存区如图一个 scull 设备的布局所示．

图 3.1. 一个 scull 设备的布局



选定的数字是这样, 在 scull 中写单个一个字节消耗 8000 或 12,000 KB 内存: 4000 是量子, 4000 或者 8000 是量子集(根据指针在目标平台上是用 32 位还是 64 位表示). 相反, 如果你写入大量数据, 链表的开销不是太坏. 每 4 MB 数据只有一个链表元素, 设备的最大尺寸受限于计算机的内存大小.

为量子和量子集选择合适的值是一个策略问题, 而不是机制, 并且优化的值依赖于设备如何使用. 因此, scull 驱动不应当强制给量子和量子集使用任何特别的值. 在 scull 中, 用户可以掌管改变这些值, 有几个途径:编译时间通过改变 scull.h 中的宏 SCULL_QUANTUM 和 SCULL_QSET, 在模块加载时设定整数值 scull_quantum 和 scull_qset, 或者使用 ioctl 在运行时改变当前值和缺省值.

使用宏定义和一个整数值来进行编译时和加载时配置, 是对于如何选择主编号的回忆. 我们在驱动中任何与策略相关或专断的值上运用这个技术.

余下的唯一问题是如果选择缺省值. 在这个特殊情况下, 问题是找到最好的平衡, 由填充了一半的量子和量子集导致内存浪费, 如果量子和量子集小的情况下分配释放和指针连接引起开销. 另外, kmalloc 的内部设计应当考虑进去. (现在我们不追求这点, 不过; kmalloc 的内部在第 8 章探索.) 缺省值的选择来自假设测试时可能有大量数据写进 scull, 尽管设备的正常使用最可能只传送几 KB 数据.

我们已经见过内部代表我们设备的 scull_dev 结构. 结构的 quantum 和 qset 分别代表设备的量子和量子集大小. 实际数据, 但是, 是由一个不同的结构跟踪, 我们称为 struct scull_qset:

```
struct scull_qset {
 void **data;
 struct scull_qset *next;
};
```

下一个代码片段展示了实际中 struct scull_dev 和 struct scull_qset 是如何被用来持有数据的. sucll_trim 函数负责释放整个数据区, 由 scull_open 在文件为写而打开时调用. 它简单地遍历列表并且释放它发现的任何量子和量子集.

```
int scull_trim(struct scull_dev *dev)
{
        struct scull_qset *next, *dptr;
        int qset = dev->qset; /* "dev" is not-null */
        int i;
        for (dptr = dev->data; dptr; dptr = next)
        { /* all the list items */
                if (dptr->data) {
                        for (i = 0; i < qset; i++)
                                kfree(dptr->data[i]);
                        kfree(dptr->data);
                        dptr->data = NULL;
                }

                next = dptr->next;
                kfree(dptr);
        }
```

```
        dev->size = 0;
        dev->quantum = scull_quantum;
        dev->qset = scull_qset;
        dev->data = NULL;
        return 0;
}
```

scull_trim 也用在模块清理函数中，来归还 scull 使用的内存给系统.

3.7. 读和写

读和写方法都进行类似的任务，就是，从和到应用程序代码拷贝数据. 因此，它们的原型相当相似，可以同时介绍它们:

```
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
```

对于 2 个方法，filp 是文件指针，count 是请求的传输数据大小. buff 参数指向持有被写入数据的缓存，或者放入新数据的空缓存. 最后，offp 是一个指针指向一个"long offset type"对象，它指出用户正在存取的文件位置. 返回值是一个"signed size type"；它的使用在后面讨论.

让我们重复一下，read 和 write 方法的 buff 参数是用户空间指针. 因此，它不能被内核代码直接解引用. 这个限制有几个理由:

- 依赖于你的驱动运行的体系，以及内核被如何配置的，用户空间指针当运行于内核模式可能根本是无效的. 可能没有那个地址的映射，或者它可能指向一些其他的随机数据.
- 就算这个指针在内核空间是同样的东西，用户空间内存是分页的，在做系统调用时这个内存可能没有在 RAM 中. 试图直接引用用户空间内存可能产生一个页面错，这是内核代码不允许做的事情. 结果可能是一个"oops"，导致进行系统调用的进程死亡.
- 置疑中的指针由一个用户程序提供，它可能是错误的或者恶意的. 如果你的驱动盲目地解引用一个用户提供的指针，它提供了一个打开的门路使用户空间程序存取或覆盖系统任何地方的内存. 如果你不想负责你的用户的系统的安全危险，你就不能直接解引用用户空间指针.

显然，你的驱动必须能够存取用户空间缓存以完成它的工作. 但是，为安全起见这个存取必须使用特殊的，内核提供的函数. 我们介绍几个这样的函数(定义于 <asm/uaccess.h>)，剩下的在第一章"使用 ioctl 参数"一节中. 它们使用一些特殊的，依赖体系的技巧来确保内核和用户空间的数据传输安全和正确.

scull 中的读写代码需要拷贝一整段数据到或者从用户地址空间. 这个能力由下列内核函数提供，它们拷贝一个任意的字节数组，并且位于大部分读写实现的核心中.

```
unsigned long copy_to_user(void __user *to,const void *from,unsigned long count);
unsigned long copy_from_user(void *to,const void __user *from,unsigned long count);
```

尽管这些函数表现象正常的 memcpy 函数，必须加一点小心在从内核代码中存取用户空间.寻址的用户也当前可能不在内存，虚拟内存子系统会使进程睡眠在这个页被传送到位时.例如，这发生在必须从交换空间获取页的时候. 对于驱动编写者来说，最终结果是任何存取用户空间的函数必须是可重入的，必须能够和其他驱动函数并行执行，并且，特别的，必须在一个它能够合法地睡眠的位置. 我们在第 5 章再回到这个主题.

这 2 个函数的角色不限于拷贝数据到和从用户空间：它们还检查用户空间指针是否有效.如果指针无效，不进行拷贝；如果在拷贝中遇到一个无效地址，另一方面，只拷贝部分数据. 在 2 种情况下，返回值是还要拷贝的数据量. scull 代码查看这个错误返回，并且如果它不是 0 就返回 -EFAULT 给用户.

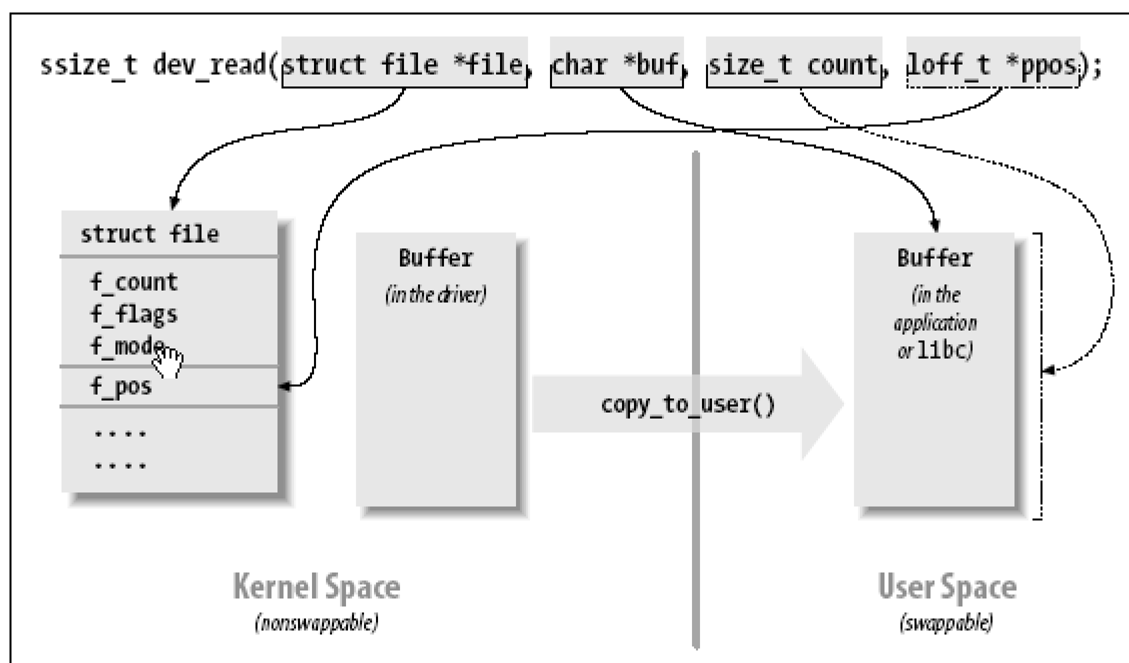
用户空间存取和无效用户空间指针的主题有些高级，在第 6 章讨论. 然而，值得注意的是如果你不需要检查用户空间指针，你可以调用 __copy_to_user 和 __copy_from_user 来代替. 这是有用处的，例如，如果你知道你已经检查了这些参数. 但是，要小心；事实上，如果你不检查你传递给这些函数的用户空间指针，那么你可能造成内核崩溃和/或安全漏洞.

至于实际的设备方法，read 方法的任务是从设备拷贝数据到用户空间(使用 copy_to_user)，而 write 方法必须从用户空间拷贝数据到设备(使用 copy_from_user).每个 read 或 write 系统调用请求一个特定数目字节的传送，但是驱动可自由传送较少数据 -- 对读和写这确切的规则稍微不同，在本章后面描述.

不管这些方法传送多少数据，它们通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置. 内核接着在适当时候传播文件位置的改变到文件结构. pread 和 pwrite 系统调用有不同的语义；它们从一个给定的文件偏移操作，并且不改变其他的系统调用看到的文件位置. 这些调用传递一个指向用户提供的位置的指针，并且放弃你的驱动所做的改变.

图给 read 的参数表示了一个典型读实现是如何使用它的参数.

图 3.2. 给 read 的参数



read 和 write 方法都在发生错误时返回一个负值. 相反, 大于或等于 0 的返回值告知调用程序有多少字节已经成功传送. 如果一些数据成功传送接着发生错误, 返回值必须是成功传送的字节数, 错误不报告直到函数下一次调用. 实现这个传统, 当然, 要求你的驱动记住错误已经发生, 以便它们可以在以后返回错误状态.

尽管内核函数返回一个负数指示一个错误, 这个数的值指出所发生的错误类型(如第 2 章介绍), 用户空间运行的程序常常看到 -1 作为错误返回值. 它们需要存取 errno 变量来找出发生了什么. 用户空间的行为由 POSIX 标准来规定, 但是这个标准没有规定内核内部如何操作.

3.7.1. read 方法

read 的返回值由调用的应用程序解释:

- 如果这个值等于传递给 read 系统调用的 count 参数, 请求的字节数已经被传送. 这是最好的情况.
- 如果是正数, 但是小于 count, 只有部分数据被传送. 这可能由于几个原因, 依赖于设备. 常常, 应用程序重新试着读取. 例如, 如果你使用 fread 函数来读取, 库函数重新发出系统调用直到请求的数据传送完成.
- 如果值为 0, 到达了文件末尾(没有读取数据).
- 一个负值表示有一个错误. 这个值指出了什么错误, 根据 <linux/errno.h>. 出错的典型返回值包括 -EINTR(被打断的系统调用) 或者 -EFAULT(坏地址).

前面列表中漏掉的是这种情况"没有数据, 但是可能后来到达". 在这种情况下, read 系统调用应当阻塞. 我们将在第 6 章涉及阻塞.

scull 代码利用了这些规则. 特别地, 它利用了部分读规则. 每个 scull_read 调用只处理单个数据量子, 不实现一个循环来收集所有的数据; 这使得代码更短更易读. 如果读程序确实需要更多数据, 它重新调用. 如果标准 I/O 库(例如, fread)用来读取设备, 应用程序甚至不会注意到数据传送的量子化.

如果当前读取位置大于设备大小, scull 的 read 方法返回 0 来表示没有可用的数据(换句话说, 我们在文件尾). 这个情况发生在如果进程 A 在读设备, 同时进程 B 打开它写, 这样将设备截短为 0. 进程 A 突然发现自己过了文件尾, 下一个读调用返回 0.

这是 read 的代码(忽略对 down_interruptible 的调用并且现在为 up; 我们在下一章中讨论它们):

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr; /* the first listitem */
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset; /* how many bytes in the listitem */
        int item, s_pos, q_pos, rest;
        ssize_t retval = 0;

        if (down_interruptible(&dev->sem))
```

```
                return -ERESTARTSYS;
        if (*f_pos >= dev->size)
                goto out;
        if (*f_pos + count > dev->size)
                count = dev->size - *f_pos;

        /* find listitem, qset index, and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum;
        q_pos = rest % quantum;

        /* follow the list up to the right position (defined elsewhere) */
        dptr = scull_follow(dev, item);
        if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
                goto out; /* don't fill holes */

        /* read only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;

        if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count))
        {
                retval = -EFAULT;
                goto out;
        }
        *f_pos += count;
        retval = count;

out:
        up(&dev->sem);
        return retval;
}
```

3.7.2. write 方法

write, 象 read, 可以传送少于要求的数据, 根据返回值的下列规则:

- 如果值等于 count, 要求的字节数已被传送.
- 如果正值, 但是小于 count, 只有部分数据被传送. 程序最可能重试写入剩下的数据.
- 如果值为 0, 什么没有写. 这个结果不是一个错误, 没有理由返回一个错误码. 再一次, 标准库重试写调用. 我们将在第 6 章查看这种情况的确切含义, 那里介绍了阻塞.
- 一个负值表示发生一个错误; 如同对于读, 有效的错误值是定义于 <linux/errno.h>中.

不幸的是, 仍然可能有发出错误消息的不当行为程序, 它在进行了部分传送时终止. 这是因为一些程序员习惯看写调用要么完全失败要么完全成功, 这实际上是大部分时间的情况, 应当也被设备支持. scull 实现的这个限制可以修改, 但是我们不想使代码不必要地复杂.

write 的 scull 代码一次处理单个量子，如 read 方法做的：

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        /* find listitem, qset index and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum;
        q_pos = rest % quantum;
        /* follow the list up to the right position */
        dptr = scull_follow(dev, item);
        if (dptr == NULL)
                goto out;
        if (!dptr->data)
        {
                dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
                if (!dptr->data)
                        goto out;
                memset(dptr->data, 0, qset * sizeof(char *));
        }
        if (!dptr->data[s_pos])
        {
                dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
                if (!dptr->data[s_pos])
                        goto out;
        }
        /* write only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;
        if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count))
        {
                retval = -EFAULT;
                goto out;
        }
        *f_pos += count;
        retval = count;

        /* update the size */
        if (dev->size < *f_pos)
                dev->size = *f_pos;
out:
        up(&dev->sem);
        return retval;
}
```

3.7.3. readv 和 writev

Unix 系统已经长时间支持名为 readv 和 writev 的 2 个系统调用. 这些 read 和 write 的"矢量"版本使用一个结构数组, 每个包含一个缓存的指针和一个长度值. 一个 readv 调用被期望来轮流读取指示的数量到每个缓存. 相反, writev 要收集每个缓存的内容到一起并且作为单个写操作送出它们.

如果你的驱动不提供方法来处理矢量操作, readv 和 writev 由多次调用你的 read 和 write 方法来实现. 在许多情况, 但是, 直接实现 readv 和 writev 能获得更大的效率.

矢量操作的原型是:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t
*ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t
*ppos);
```

这里, filp 和 ppos 参数与 read 和 write 的相同. iovec 结构, 定义于 <linux/uio.h>, 如同:

```
struct iovec
{
    void __user *iov_base; __kernel_size_t iov_len;
};
```

每个 iovec 描述了一块要传送的数据; 它开始于 iov_base (在用户空间)并且有 iov_len 字节长. count 参数告诉有多少 iovec 结构. 这些结构由应用程序创建, 但是内核在调用驱动之前拷贝它们到内核空间.

矢量操作的最简单实现是一个直接的循环, 只是传递出去每个 iovec 的地址和长度给驱动的 read 和 write 函数. 然而, 有效的和正确的行为常常需要驱动更聪明. 例如, 一个磁带驱动上的 writev 应当将全部 iovec 结构中的内容作为磁带上的单个记录.

很多驱动, 但是, 没有从自己实现这些方法中获益. 因此, scull 省略它们. 内核使用 read 和 write 来模拟它们, 最终结果是相同的.

3.8. 使用新设备

一旦你装备好刚刚描述的 4 个方法, 驱动可以编译并测试了; 它保留了你写给它的任何数据, 直到你用新数据覆盖它. 这个设备表现如一个数据缓存器, 它的长度仅仅受限于可用的真实 RAM 的数量. 你可试着使用 cp, dd, 以及 输入/输出重定向来测试这个驱动.

free 命令可用来看空闲内存的数量如何缩短和扩张的, 依据有多少数据写入 scull.

为对一次读写一个量子有更多信心, 你可增加一个 printk 在驱动的适当位置, 并且观察当应用程序读写大块数据中发生了什么. 可选地, 使用 strace 工具来监视程序发出的系统调用以及它们的返回值. 跟踪一个 cp 或者一个 ls -l > /dev/scull0 展示了量子化的读和写. 监视(以及调试)技术在第 4 章详细介绍.

3.9. 快速参考

本章介绍了下面符号和头文件. struct file_operations 和 struct file 中的成员的列表这里不重复了.

```
#include <linux/types.h>
dev_t
```

dev_t 是用来在内核里代表设备号的类型.

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

从设备编号中抽取主次编号的宏.

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

从主次编号来建立 dev_t 数据项的宏定义.

```
#include <linux/fs.h>
```

"文件系统"头文件是编写设备驱动需要的头文件. 许多重要的函数和数据结构在此定义.

```
int register_chrdev_region(dev_t first, unsigned int count, char *name)
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char
*name)
void unregister_chrdev_region(dev_t first, unsigned int count);
```

允许驱动分配和释放设备编号的范围的函数. register_chrdev_region 应当用在事先知道需要的主编号时; 对于动态分配, 使用 alloc_chrdev_region 代替.

```
int register_chrdev(unsigned int major, const char *name, struct
file_operations *fops);
```

老的(2.6 之前) 字符设备注册函数. 它在 2.6 内核中被模拟, 但是不应当给新代码使用. 如果主编号不是 0, 可以不变地用它; 否则一个动态编号被分配给这个设备.

```
int unregister_chrdev(unsigned int major, const char *name);
```

恢复一个由 register_chrdev 所作的注册的函数. major 和 name 字符串必须包含之前用来注册设备时同样的值.

```
struct file_operations;
struct file;
struct inode;
```

大部分设备驱动使用的 3 个重要数据结构. file_operations 结构持有一个字符驱动的方法; struct file 代表一个打开的文件, struct inode 代表磁盘上的一个文件.

```
#include <linux/cdev.h>
struct cdev *cdev_alloc(void);
void cdev_init(struct cdev *dev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
void cdev_del(struct cdev *dev);
```

cdev 结构管理的函数, 它代表内核中的字符设备.

```
#include <linux/kernel.h>
container_of(pointer, type, field);
```

一个传统宏定义, 可用来获取一个结构指针, 从它里面包含的某个其他结构的指针.

```
#include <asm/uaccess.h>
```

这个包含文件声明内核代码使用的函数来移动数据到和从用户空间.

```
unsigned long copy_from_user (void *to, const void *from, unsigned long count);
unsigned long copy_to_user (void *to, const void *from, unsigned long count);
```

在用户空间和内核空间拷贝数据.

第 4 章 调试技术

内核编程带有它自己的，独特的调试挑战性. 内核代码无法轻易地在一个调试器下运行，也无法轻易的被跟踪，因为它是一套没有与特定进程相关连的功能的集合. 内核代码错误也特别难以重现，它们会牵连整个系统与它们一起失效，从而破坏了大量的能用来追踪错误的证据.

本章介绍了在如此艰难情况下能够用以监视内核代码和跟踪错误的技术.

4.1. 内核中的调试支持

在第 2 章，我们建议你建立并安装你自己的内核，而不是运行来自你的发布商的现成的内核. 运行你自己的内核的最充分的理由之一是内核开发者已经在内核自身中构建了多个调试特性. 这些特性能产生额外的输出并降低性能，因此发布商的产品内核中往往不会使能它们. 作为一个内核开发者，但是，你有不同的优先权并且会乐于接收这些格外的内核调试支持带来的开销.

这里，我们列出用来开发的内核应当激活的配置选项. 除了另外指出的，所有的这些选项都在 "kernel hacking" 菜单，不管什么样的你喜欢的内核配置工具. 注意有些选项不是所有体系都支持.

CONFIG_DEBUG_KERNEL

这个选项只是使其他调试选项可用；它应当打开，但是它自己不激活任何的特性.

CONFIG_DEBUG_SLAB

这个重要的选项打开了内核内存分配函数的几类检查；激活这些检查，就可能探测到一些内存覆盖和遗漏初始化的错误. 被分配的每一个字节在递交给调用者之前都设成 0xa5，随后在释放时被设成 0x6b. 你在任何时候如果见到任一个这种"坏"模式重复出现在你的驱动输出(或者常常在一个 oops 的列表)，你会确切知道去找什么类型的错误. 当激活调试，内核还会在每个分配的内存对象的前后放置特别的守护值；如果这些值曾被改动，内核知道有人已覆盖了一个内存分配区，它大声抱怨. 各种的对更模糊的问题的检查也给激活了.

CONFIG_DEBUG_PAGEALLOC

满的页在释放时被从内核地址空间去除. 这个选项会显著拖慢系统，但是它也能快速指出某些类型的内存损坏错误.

CONFIG_DEBUG_SPINLOCK

激活这个选项，内核捕捉对未初始化的自旋锁的操作，以及各种其他的错误(例如 2 次解锁同一个锁).

CONFIG_DEBUG_SPINLOCK_SLEEP

这个选项激活对持有自旋锁时进入睡眠的检查. 实际上, 如果你调用一个可能会睡眠的函数, 它就抱怨, 即便这个有疑问的调用没有睡眠.

CONFIG_INIT_DEBUG

用__init (或者 __initdata) 标志的项在系统初始化或者模块加载后都被丢弃. 这个选项激活了对代码的检查, 这些代码试图在初始化完成后存取初始化时内存.

CONFIG_DEBUG_INFO

这个选项使得内核在建立时包含完整的调试信息. 如果你想使用 gdb 调试内核, 你将需要这些信息. 如果你打算使用 gdb, 你还要激活 CONFIG_FRAME_POINTER.

CONFIG_MAGIC_SYSRQ

激活"魔术 SysRq"键. 我们在本章后面的"系统挂起"一节查看这个键.

CONFIG_DEBUG_STACKOVERFLOW
CONFIG_DEBUG_STACK_USAGE

这些选项能帮助跟踪内核堆栈溢出. 堆栈溢出的确证是一个 oops 输出, 但是没有任何形式的合理的回溯. 第一个选项给内核增加了明确的溢出检查; 第 2 个使得内核监测堆栈使用并作一些统计, 这些统计可以用魔术 SysRq 键得到.

CONFIG_KALLSYMS

这个选项(在"Generl setup/Standard features"下)使得内核符号信息建在内核中; 缺省是激活的. 符号选项用在调试上下文中; 没有它, 一个 oops 列表只能以 16 进制格式给你一个内核回溯, 这不是很有用.

CONFIG_IKCONFIG
CONFIG_IKCONFIG_PROC

这些选项(在"Generl setup"菜单)使得完整的内核配置状态被建立到内核中, 可以通过 /proc 来使其可用. 大部分内核开发者知道他们使用的哪个配置, 并不需要这些选项(会使得内核更大). 但是如果你试着调试由其他人建立的内核中的问题, 它们可能有用.

CONFIG_ACPI_DEBUG

在"Power management/ACPI"下. 这个选项打开详细的 ACPI (Advanced Configuration and Power Interface) 调试信息, 它可能有用如果你怀疑一个问题和 ACPI 相关.

CONFIG_DEBUG_DRIVER

在"Device drivers"下. 打开了驱动核心的调试信息, 可用以追踪低层支持代码的问题. 我们在第 14 章查看驱动核心.

CONFIG_SCSI_CONSTANTS

这个选项，在"Device drivers/SCSI device support"下，建立详细的 SCSI 错误消息的信息. 如果你在使用 SCSI 驱动，你可能需要这个选项.

CONFIG_INPUT_EVBUG

这个选项(在"Device drivers/Input device support"下)打开输入事件的详细日志. 如果你使用一个输入设备的驱动，这个选项可能会有用. 然而要小心这个选项的安全性的隐含意义：它记录了你键入的任何东西，包括你的密码.

CONFIG_PROFILING

这个选项位于"Profiling support"之下. 剖析通常用在系统性能调整，但是在追踪一些内核挂起和相关问题上也有用.

我们会再次遇到一些上面的选项，当我们查看各种方法来追踪内核问题时. 但是首先，我们要看一下经典的调试技术：print 语句.

4.2. 用打印调试

最常用的调试技术是监视，在应用程序编程当中是通过在合适的地方调用 printf 来实现. 在你调试内核代码时，你可以通过 printk 来达到这个目的.

4.2.1. printk

我们在前面几章中使用 printk 函数，简单地假设它如同 printf 一样使用. 现在到时候介绍一些不同的地方了.

一个不同是 printk 允许你根据消息的严重程度对其分类，通过附加不同的记录级别或者优先级在消息上. 你常常用一个宏定义来指示记录级别. 例如，KERN_INFO，我们之前曾在一些打印语句的前面看到过，是消息记录级别的一种可能值. 记录宏定义扩展成一个字串，在编译时与消息文本连接在一起；这就是为什么下面的在优先级和格式串之间没有逗号的原因. 这里有 2 个 printk 命令的例子，一个调试消息，一个紧急消息：

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

有 8 种可能的记录字串，在头文件 <linux/kernel.h> 里定义；我们按照严重性递减的顺序列出它们：

KERN_EMERG

用于紧急消息，常常是那些崩溃前的消息.

KERN_ALERT

需要立刻动作的情形.

KERN_CRIT

严重情况，常常与严重的硬件或者软件失效有关.

KERN_ERR

用来报告错误情况；设备驱动常常使用 KERN_ERR 来报告硬件故障.

KERN_WARNING

有问题的情况的警告，这些情况自己不会引起系统的严重问题.

KERN_NOTICE

正常情况，但是仍然值得注意. 在这个级别一些安全相关的情况会报告.

KERN_INFO

信息型消息. 在这个级别，很多驱动在启动时打印它们发现的硬件的信息.

KERN_DEBUG

用作调试消息.

每个字串(在宏定义扩展里)代表一个在角括号中的整数. 整数的范围从 0 到 7，越小的数表示越大的优先级.

一条没有指定优先级的 printk 语句缺省是 DEFAULT_MESSAGE_LOGLEVEL，在 kernel/printk.c 里指定作为一个整数. 在 2.6.10 内核中，DEFAULT_MESSAGE_LOGLEVEL 是 KERN_WARNING，但是在过去已知是改变的.

基于记录级别，内核可能打印消息到当前控制台，可能是一个文本模式终端，串口，或者是一台并口打印机. 如果优先级小于整型值 console_loglevel，消息被递交给控制台，一次一行(除非提供一个新行结尾，否则什么都不发送). 如果 klogd 和 syslogd 都在系统中运行，内核消息被追加到 /var/log/messages (或者另外根据你的 syslogd 配置处理)，独立于 console_loglevel. 如果 klogd 没有运行，你只有读 /proc/kmsg (用 dmsg 命令最易做到)将消息取到用户空间. 当使用 klogd 时，你应当记住，它不会保存连续的同样的行；它只保留第一个这样的行，随后是，它收到的重复行数.

变量 console_loglevel 初始化成 DEFAULT_CONSOLE_LOGLEVEL，并且可通过 sys_syslog 系统调用修改. 一种修改它的方法是在调用 klogd 时指定 -c 开关，在 klogd 的 manpage 里有指定. 注意要改变当前值，你必须先杀掉 klogd，接着使用 -c 选项重启它. 另外，你可写一个程序来改变控制台记录级别. 你会发现这样一个程序的版本在由 O' Reilly 提供的 FTP 站点上的 miscprogs/setlevel.c. 新的级别指定未一个整数，在 1 和 8 之前，包含 1 和 8. 如果它设为 1，只有 0 级消息(KERN_EMERG)到达控制台；如果它设为 8，所有消息，包括调试消息，都显示.

也可以通过文本文件 /proc/sys/kernel/printk 读写控制台记录级别. 这个文件有 4 个整型值: 当前记录级别, 适用没有明确记录级别的消息的缺省级别, 允许的最小记录级别, 以及启动时缺省记录级别. 写一个单个值到这个文件就改变当前记录级别成这个值; 因此, 例如, 你可以使所有内核消息出现在控制台, 通过简单地输入:

```
 # echo 8 > /proc/sys/kernel/printk
```

现在应当清楚了为什么 hello.c 例子使用 KERN_ALERT 标志; 它们是要确保消息会出现在控制台上.

4.2.2. 重定向控制台消息

Linux 在控制台记录策略上允许一些灵活性, 它允许你发送消息到一个指定的虚拟控制台(如果你的控制台使用的是文本屏幕). 缺省地, 这个"控制台"是当前虚拟终端. 为了选择一个不同地虚拟终端来接收消息, 你可对任何控制台设备调用 ioctl(TIOCLINUX). 下面的程序, setconsole, 可以用来选择哪个控制台接收内核消息; 它必须由超级用户运行, 可以从 misc-progs 目录得到.

下面是全部程序. 应当使用一个参数来指定用以接收消息的控制台的编号.

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11 is the TIOCLINUX cmd number */
    if (argc==2) bytes[1] = atoi(argv[1]); /* the chosen console */
    else {

        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1); } if
(ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) { /* use stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
                argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

setconsole 使用特殊的 ioctl 命令 TIOCLINUX, 来实现特定于 linux 的功能. 为使用 TIOCLINUX, 你传递它一个指向字节数组的指针作为参数. 数组的第一个字节是一个数, 指定需要的子命令, 下面的字节是特对于子命令的. 在 setconsole 里, 使用子命令 11, 下一个字节(存于 bytes[1])指定虚拟控制台. TIOCLINUX 的完整描述在内核源码的 drivers/char/tty_io.c 里.

4.2.3. 消息是如何记录的

printk 函数将消息写入一个 __LOG_BUF_LEN 字节长的环形缓存, 长度值从 4 KB 到 1 MB, 由配置内核时选择. 这个函数接着唤醒任何在等待消息的进程, 就是说, 任何在系统调用中睡眠或者在读取 /proc/kmsg 的进程. 这 2 个日志引擎的接口几乎是等同的, 但是注意, 从 /proc/kmsg 中读取是从日志缓存中消费数据, 然而 syslog 系统调用能够选择地在返回日志数据地同时保留它给其他进程. 通常, 读取 /proc 文件容易些并且是

klogd 的缺省做法. dmesg 命令可用来查看缓存的内容，不会冲掉它；实际上，这个命令将缓存区的整个内容返回给 stdout，不管它是否已经被读过.

在停止 klogd 后，如果你偶尔手工读取内核消息，你会发现 /proc 看起来象一个 FIFO，读者阻塞在里面，等待更多数据. 显然，你无法以这种方式读消息，如果 klogd 或者其他进程已经在读同样的数据，因为你要竞争它.

如果环形缓存填满，printk 绕回并在缓存的开头增加新数据，覆盖掉最老的数据. 因此，这个记录过程会丢失最老的数据. 这个问题相比于使用这样一个环形缓存的优点是可以忽略的. 例如，环形缓存允许系统即便没有一个日志进程也可运行，在没有人读它的时候可以通过覆盖旧数据浪费最少的内存. Linux 对于消息的解决方法的另一个特性是，printk 可以从任何地方调用，甚至从一个中断处理里面，没有限制能打印多少数据. 唯一的缺点是可能丢失一些数据.

如果 klogd 进程在运行，它获取内核消息并分发给 syslogd，syslogd 接着检查 /etc/syslog.conf 来找出如何处理它们. syslogd 根据一个设施和一个优先级来区分消息；这个设施和优先级的允许值在 <sys/syslog.h> 中定义. 内核消息由 LOG_KERN 设施来记录，在一个对应于 printk 使用的优先级上(例如，LOG_ERR 用于 KERN_ERR 消息). 如果 klogd 没有运行，数据保留在环形缓存中直到有人读它或者缓存被覆盖.

如果你要避免你的系统被来自你的驱动的监视消息击垮，你或者给 klogd 指定一个 -f (文件) 选项来指示它保存消息到一个特定的文件，或者定制 /etc/syslog.conf 来适应你的要求. 但是另外一种可能性是采用粗暴的方式：杀掉 klogd 和详细地打印消息在一个没有用到的虚拟终端上，[13] [13] 或者从一个没有用到的 xterm 上发出命令 cat /proc/kmsg.

4.2.4. 打开和关闭消息

在驱动开发的早期，printk 非常有助于调试和测试新代码. 当你正式发行驱动时，换句话说，你应当去掉，或者至少关闭，这些打印语句. 不幸的是，你很可能会发现，就在你认为你不再需要这些消息并去掉它们时，你要在驱动中实现一个新特性(或者有人发现了一个 bug)，你想要至少再打开一个消息. 有几个方法来解决这 2 个问题，全局性地打开或关闭你地调试消息和打开或关闭单个消息.

这里我们展示一种编码 printk 调用的方法，你可以单独或全局地打开或关闭它们；这个技术依靠定义一个宏，在你想使用它时就转变成一个 printk (或者 printf)调用.

- 每个 printk 语句可以打开或关闭，通过去除或添加单个字符到宏定义的名子.
- 所有消息可以马上关闭，通过在编译前改变 CFLAGS 变量的值.
- 同一个 print 语句可以在内核代码和用户级代码中使用，因此对于格外的消息，驱动和测试程序能以同样的方式被管理.

下面的代码片断实现了这些特性，直接来自头文件 scull.h:

[13] [13] * 例如，使用 setlevel 8; setconsole 10 来配置终端 10 来显示消息.

```
#undef PDEBUG /* undef it, just in case */
#ifdef SCULL_DEBUG
# ifdef __KERNEL__

/* This one if debugging is on, and kernel space */
# define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
# else

/* This one for user space */
# define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG #define PDEBUGG(fmt, args...)        /* nothing: it's a placeholder */
```

符号 PDEBUG 定义和去定义，取决于 SCULL_DEBUG 是否定义，和以何种方式显示消息适合代码运行的环境：当它在内核中就使用内核调用 printk，在用户空间运行就使用 libc 调用 fprintf 到标准错误输出. PDEBUGG 符号，换句话说，什么不作；他可用来轻易地"注释" print 语句，而不用完全去掉它们.

为进一步简化过程，添加下面的行到你的 makfile 里：

```
# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
 DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
 DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

本节中出现的宏定义依赖 gcc 对 ANSI C 预处理器的扩展，支持带可变个数参数的宏定义. 这个 gcc 依赖不应该是个问题，因为无论如何内核固有的非常依赖于 gcc 特性. 另外，makefile 依赖 GNU 版本的 make；再一次，内核也依赖 GNU make，所以这个依赖不是问题.

如果你熟悉 C 预处理器，你可以扩展给定的定义来实现一个"调试级别"的概念，定义不同的级别，安排一个整数(或者位掩码)值给每个级别，以便决定它应当多么详细.

但是每个驱动有它自己的特性和监视需求. 好的编程技巧是在灵活性和效率之间选择最好的平衡，我们无法告诉你什么是最好的. 记住，预处理器条件(连同代码中的常数表达式)在编译时执行，因此你必须重新编译来打开或改变消息. 一个可能的选择是使用 C 条件句，它在运行时执行，因而，能允许你在出现执行时打开或改变消息机制. 这是一个好的特性，但是它在每次代码执行时需要额外的处理，这样即便消息给关闭了也会影响效率. 有时这个效率损失无法接受.

本节出现的宏定义已经证明在多种情况下是有用的，唯一的缺点是要求在任何对它的消息改变后重新编译.

4.2.5. 速率限制

如果你不小心，你会发现自己用 printk 产生了上千条消息，压倒了控制台并且，可能地，使系统日志文件溢出. 当使用一个慢速控制台设备(例如，一个串口)，过量的消息速率也能拖慢系统或者只是使它不反应了. 非常难于着手于系统出错的地方，当控制台不停地输出数据. 因此，你应当非常注意你打印什么，特别在驱动的产品版本以及特别在初始化完成后. 通常，产品代码在正常操作时不应当打印任何东西；打印的输出应当是指示需要注意的异常情况.

另一方面，你可能想发出一个日志消息，如果你驱动的设备停止工作. 但是你应当小心不要做过了头. 一个面对失败永远继续的傻瓜进程能产生每秒上千次的尝试；如果你的驱动每次都打印"my device is broken"，它可能产生大量的输出，如果控制台设备慢就有可能霸占 CPU -- 没有中断用来驱动控制台，就算是一个串口或者一个行打印机.

在很多情况下，最好的做法是设置一个标志说，"我已经抱怨过这个了"，并不打印任何后来的消息只要这个标志设置着. 然而，有几个理由偶尔发出一个"设备还是坏的"的提示. 内核已经提供了一个函数帮助这个情况：

```
int printk_ratelimit(void);
```

这个函数应当在你认为打印一个可能会常常重复的消息之前调用. 如果这个函数返回非零值，继续打印你的消息，否则跳过它. 这样，典型的调用如这样：

```
if (printk_ratelimit())
    printk(KERN_NOTICE "The printer is still on fire\n");
```

printk_ratelimit 通过跟踪多少消息发向控制台而工作. 当输出级别超过一个限度，printk_ratelimit 开始返回 0 并使消息被扔掉.

printk_ratelimit 的行为可以通过修改 /proc/sys/kern/printk_ratelimit(在重新使能消息前等待的秒数) 和 /proc/sys/kernel/printk_ratelimit_burst(限速前可接收的消息数)来定制.

4.2.6. 打印设备编号

偶尔地，当从一个驱动打印消息，你会想打印与感兴趣的硬件相关联的设备号. 打印主次编号不是特别难，但是，为一致性考虑，内核提供了一些实用的宏定义(在 <linux/kdev_t.h> 中定义)用于这个目的：

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

两个宏定义都将设备号编码进给定的缓冲区；唯一的区别是 print_dev_t 返回打印的字符数，而 format_dev_t 返回缓存区；因此，它可以直接用作 printk 调用的参数，但是

必须记住 printk 只有提供一个结尾的新行才会刷行. 缓冲区应当足够大以存放一个设备号; 如果 64 位编号在以后的内核发行中明显可能, 这个缓冲区应当可能至少是 20 字节长.

4.3. 用查询来调试

前面一节描述了 printk 是任何工作的以及怎样使用它. 没有谈到的是它的缺点.

大量使用 printk 能够显著地拖慢系统, 即便你降低 cosole_loglevel 来避免加载控制台设备, 因为 syslogd 会不停地同步它的输出文件; 因此, 要打印的每一行都引起一次磁盘操作. 从 syslogd 的角度这是正确的实现. 它试图将所有东西写到磁盘上, 防止系统刚好在打印消息后崩溃; 然而, 你不想只是为了调试信息的原因而拖慢你的系统. 可以在出现于 /etc/syslogd.conf 中的你的日志文件名前加一个连字号来解决这个问题[14][14]. 改变配置文件带来的问题是, 这个改变可能在你结束调试后保留在那里, 即便在正常系统操作中你确实想尽快刷新消息到磁盘. 这样永久改变的另外的选择是运行一个非 klogd 程序(例如 cat /proc/kmsg, 如之前建议的), 但是这可能不会提供一个合适的环境给正常的系统操作.

经常地, 最好的获得相关信息的方法是查询系统, 在你需要消息时, 不是连续地产生数据. 实际上, 每个 Unix 系统提供许多工具来获取系统消息: ps, netstat, vmstat, 等等.

有几个技术给驱动开发者来查询系统: 创建一个文件在 /proc 文件系统下, 使用 ioctl 驱动方法, 借助 sysfs 输出属性. 使用 sysfs 需要不少关于驱动模型的背景知识. 在 14 章讨论.

4.3.1. 使用 /proc 文件系统

/proc 文件系统是一个特殊的软件创建的文件系统, 内核用来输出消息到外界. /proc 下的每个文件都绑到一个内核函数上, 当文件被读的时候即时产生文件内容. 我们已经见到一些这样的文件起作用; 例如, /proc/modules, 常常返回当前已加载的模块列表.

/proc 在 Linux 系统中非常多地应用. 很多现代 Linux 发布中的工具, 例如 ps, top, 以及 uptime, 从 /proc 中获取它们的信息. 一些设备驱动也通过 /proc 输出信息, 你的也可以这样做. /proc 文件系统是动态的, 因此你的模块可以在任何时候添加或去除条目.

完全特性的 /proc 条目可能是复杂的野兽; 另外, 它们可写也可读, 但是, 大部分时间, /proc 条目是只读的文件. 本节只涉及简单的只读情况. 那些感兴趣于实现更复杂的东西的人可以从这里获取基本知识; 接下来可参考内核源码来获知完整的信息.

在我们继续之前, 我们应当提及在 /proc 下添加文件是不鼓励的. /proc 文件系统在内核开发者看作是有点无法控制的混乱, 它已经远离它的本来目的了(是提供关于系统中运行的进程的信息). 建议新代码中使信息可获取的方法是利用 sysfs. 如同建议的, 使用

[14] [14] 连字号, 或者减号, 是一个"魔术"标识以阻止 syslogd 刷新文件到磁盘在每个新消息, 有关文档在 syslog.conf(5), 一个值得一读的 manpage.

sysfs 需要对 Linux 设备模型的理解，然而，我们直到 14 章才接触它. 同时，/proc 下的文件稍稍容易创建，并且它们完全适合调试目的，所以我们在这里包含它们.

4.3.1.1. 在 /proc 里实现文件

所有使用 /proc 的模块应当包含 <linux/proc_fs.h> 来定义正确的函数.

要创建一个只读 /proc 文件，你的驱动必须实现一个函数来在文件被读时产生数据. 当某个进程读文件时(使用 read 系统调用)，这个请求通过这个函数到达你的模块. 我们先看看这个函数并在本章后面讨论注册接口.

当一个进程读你的 /proc 文件，内核分配了一页内存(就是说，PAGE_SIZE 字节)，驱动可以写入数据来返回给用户空间. 那个缓存区传递给你的函数，是一个称为 read_proc 的方法:

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof, void *data);
```

page 指针是你写你的数据的缓存区；start 是这个函数用来说有关的数据写在页中哪里(下面更多关于这个)；offset 和 count 对于 read 方法有同样的含义. eof 参数指向一个整数，必须由驱动设置来指示它不再有数据返回，data 是驱动特定的数据指针，你可以用做内部用途.

这个函数应当返回实际摆放于 page 缓存区的数据的字节数，就象 read 方法对别的文件所作一样. 别的输出值是 *eof 和 *start. eof 是一个简单的标志，但是 start 值的使用有些复杂；它的目的是帮助实现大的(超过一页) /proc 文件.

start 参数有些非传统的用法. 它的目的是指示哪里(哪一页)找到返回给用户的数据. 当调用你的 proc_read 方法，*start 将会是 NULL. 如果你保持它为 NULL，内核假定数据已放进 page 偏移是 0；换句话说，它假定一个头脑简单的 proc_read 版本，它安放虚拟文件的整个内容到 page，没有注意 offset 参数. 如果，相反，你设置 *start 为一个 非 NULL 值，内核认为由 *start 指向的数据考虑了 offset，并且准备好直接返回给用户. 通常，返回少量数据的简单 proc_read 方法只是忽略 start. 更复杂的方法设置 *start 为 page 并且只从请求的 offset 那里开始安放数据.

还有一段距离到 /proc 文件的另一个主要问题，它也打算解答 start. 有时内核数据结构的 ASCII 表示在连续的 read 调用中改变，因此读进程可能发现从一个调用到下一个有不一致的数据. 如果 *start 设成一个小的整数值，调用者用它来递增 filp-<f_pos 不依赖你返回的数据量，因此使 f_pos 成为你的 read_proc 过程的一个内部记录数. 如果，例如，如果你的 read_proc 函数从一个大结构数组返回信息并且第一次调用返回了 5 个结构，*start 可设成 5. 下一个调用提供同一个数作为 offset；驱动就知道从数组中第 6 个结构返回数据. 这是被它的作者承认的一个" hack "，可以在 fs/proc/generic.c 见到.

注意，有更好的方法实现大的 /proc 文件；它称为 seq_file，我们很快会讨论它. 首先，然而，是时间举个例子了. 下面是一个简单的(有点丑陋) read_proc 实现，为 scull 设备:

```
int scull_read_procmem(char *buf, char **start, off_t offset, int count, int *eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* Don't print more than this */

    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n", i, d->qset, d->quantum, d->size);
        for (; qs && len <= limit; qs = qs->next) { /* scan the list */
            len += sprintf(buf + len, " item at %p, qset at %p\n", qs, qs->data);
            if (qs->data && !qs->next) /* dump only the last item */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        len += sprintf(buf + len, " % 4i: %8p\n", j, qs->data[j]);
                }
        }
        up(&scull_devices[i].sem);

    }
    *eof = 1;
    return len;

}
```

这是一个相当典型的 read_proc 实现. 它假定不会有必要产生超过一页数据并且因此忽略了 start 和 offset 值. 它是, 但是, 小心地不覆盖它的缓存, 只是以防万一.

4.3.1.2. 老接口

如果你阅览内核源码, 你会遇到使用老接口实现 /proc 的代码:

```
int (*get_info)(char *page, char **start, off_t offset, int count);
```

所有的参数的含义同 read_proc 的相同, 但是没有 eof 和 data 参数. 这个接口仍然支持, 但是将来会消失; 新代码应当使用 read_proc 接口来代替.

4.3.1.3. 创建你的 /proc 文件

一旦你有一个定义好的 read_proc 函数, 你应当连接它到 /proc 层次中的一个入口项. 使用一个 creat_proc_read_entry 调用:

```
struct proc_dir_entry *create_proc_read_entry(const char *name,mode_t mode,
struct proc_dir_entry *base, read_proc_t *read_proc, void *data);
```

这里, name 是要创建的文件名子, mod 是文件的保护掩码(缺省系统范围时可以作为 0 传递), base 指出要创建的文件的目录(如果 base 是 NULL, 文件在 /proc 根下创建), read_proc 是实现文件的 read_proc 函数, data 被内核忽略(但是传递给 read_proc). 这就是 scull 使用的调用, 来使它的 /proc 函数可用做 /proc/scullmem:

```
create_proc_read_entry("scullmem", 0 /* default mode */,
                       NULL /* parent dir */, scull_read_procmem,
                       NULL /* client data */);
```

这里，我们创建了一个名为 scullmem 的文件，直接在 /proc 下，带有缺省的，全局可读的保护.

目录入口指针可用来在 /proc 下创建整个目录层次. 但是，注意，一个入口放在 /proc 的子目录下会更容易，通过简单地给出目录名子作为这个入口名子的一部分 -- 只要这个目录自身已经存在. 例如，一个(常常被忽略)传统的是 /proc 中与设备驱动相连的入口应当在 driver/ 子目录下; scull 能够安放它的入口在那里，简单地通过指定它为名子 driver/scullmem.

/proc 中的入口，当然，应当在模块卸载后去除. remove_proc_entry 是恢复 create_proc_read_entry 所做的事情的函数:

```
remove_proc_entry("scullmem", NULL /* parent dir */);
```

去除入口失败会导致在不希望的时间调用，或者，如果你的模块已被卸载，内核崩掉.

当如展示的使用 /proc 文件，你必须记住几个实现的麻烦事 -- 不要奇怪现在不鼓励使用它.

最重要的问题是关于去除 /proc 入口. 这样的去除很可能在文件使用时发生，因为没有所有者关联到 /proc 入口，因此使用它们不会作用到模块的引用计数. 这个问题可以简单的触发，例如通过运行 sleep 100 < /proc/myfile，刚好在去除模块之前.

另外一个问题时关于用同样的名子注册两个入口. 内核信任驱动，不会检查名子是否已经注册了，因此如果你不小心，你可能会使用同样的名子注册两个或多个入口. 这是一个已知发生在教室中的问题，这样的入口是不能区分的，不但在你存取它们时，而且在你调用 remove_proc_entry 时.

4.3.1.4. seq_file 接口

如我们上面提到的，在 /proc 下的大文件的实现有点麻烦. 一直以来，/proc 方法因为当输出数量变大时的错误实现变得声名狼藉. 作为一种清理 /proc 代码以及使内核开发者活得轻松些的方法，添加了 seq_file 接口. 这个接口提供了简单的一套函数来实现大内核虚拟文件.

set_file 接口假定你在创建一个虚拟文件，它涉及一系列的必须返回给用户空间的项. 为使用 seq_file，你必须创建一个简单的 "iterator" 对象，它能在序列里建立一个位置，向前进，并且输出序列里的一个项. 它可能听起来复杂，但是，实际上，过程非常简单. 我们一步步来创建 /proc 文件在 scull 驱动里，来展示它是如何做的.

第一步，不可避免地，是包含 <linux/seq_file.h>. 接着你必须创建 4 个 iterator 方法，称为 start，next，stop，和 show.

start 方法一直是首先调用. 这个函数的原型是:

```
void *start(struct seq_file *sfile, loff_t *pos);
```

sfile 参数可以几乎是一直被忽略. pos 是一个整型位置值, 指示应当从哪里读. 位置的解释完全取决于实现; 在结果文件里不需要是一个字节位置. 因为 seq_file 实现典型地步进一系列感兴趣的项, position 常常被解释为指向序列中下一个项的指针. scull 驱动解释每个设备作为系列中的一项, 因此进入的 pos 简单地是一个 scull_device 数组的索引. 因此, scull 使用的 start 方法是:

```
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
    if (*pos >= scull_nr_devs)
        return NULL;   /* No more to read */
    return scull_devices + *pos;
}
```

返回值, 如果非 NULL, 是一个可以被 iterator 实现使用的私有值.

next 函数应当移动 iterator 到下一个位置, 如果序列里什么都没有剩下就返回 NULL. 这个方法的原型是:

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

这里, v 是从前一个对 start 或者 next 的调用返回的 iterator, pos 是文件的当前位置. next 应当递增有 pos 指向的值; 根据你的 iterator 是如何工作的, 你可能(尽管可能不会)需要递增 pos 不止是 1. 这是 scull 所做的:

```
static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    (*pos)++;
    if (*pos >= scull_nr_devs)
        return NULL;
    return scull_devices + *pos;
}
```

当内核处理完 iterator, 它调用 stop 来清理:

```
void stop(struct seq_file *sfile, void *v);
```

scull 实现没有清理工作要做, 所以它的 stop 方法是空的.

设计上, 值得注意 seq_file 代码在调用 start 和 stop 之间不睡眠或者进行其他非原子性任务. 你也肯定会看到在调用 start 后马上有一个 stop 调用. 因此, 对你的 start 方法来说请求信号量或自旋锁是安全的. 只要你的其他 seq_file 方法是原子的, 调用的整个序列是原子的. (如果这一段对你没有意义, 在你读了下一章后再回到这.)

在这些调用中, 内核调用 show 方法来真正输出有用的东西给用户空间. 这个方法的原型是:

```
int show(struct seq_file *sfile, void *v);
```

这个方法应当创建序列中由 iterator v 指示的项的输出. 不应当使用 printk, 但是; 有一套特殊的用作 seq_file 输出的函数:

```
int seq_printf(struct seq_file *sfile, const char *fmt, ...);
```

这是给 seq_file 实现的 printf 对等体; 它采用常用的格式串和附加值参数. 你必须也将给 show 函数的 set_file 结构传递给它, 然而. 如果 seq_printf 返回非零值, 意思是缓存区已填充, 输出被丢弃. 大部分实现忽略了返回值, 但是.

```
int seq_putc(struct seq_file *sfile, char c);
int seq_puts(struct seq_file *sfile, const char *s);
```

它们是用户空间 putc 和 puts 函数的对等体.

```
int seq_escape(struct seq_file *m, const char *s, const char *esc);
```

这个函数是 seq_puts 的对等体, 除了 s 中的任何也在 esc 中出现的字符以八进制格式打印. esc 的一个通用值是"\t\n\\", 它使内嵌的空格不会搞乱输出和可能搞乱 shell 脚本.

```
int seq_path(struct seq_file *sfile, struct vfsmount *m, struct dentry *dentry, char *esc);
```

这个函数能够用来输出和给定命令项关联的文件名子. 它在设备驱动中不可能有用; 我们是为了完整在此包含它.

回到我们的例子; 在 scull 使用的 show 方法是:

```
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;

    if (down_interruptible (&dev->sem))
        return -ERESTARTSYS;

    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
               (int) (dev - scull_devices), dev->qset,
               dev->quantum, dev->size);

    for (d = dev->data; d; d = d->next) { /* scan the list */
        seq_printf(s, " item at %p, qset at %p\n", d, d->data);
```

```
        if (d->data && !d->next) /* dump only the last item */

            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, " % 4i: %8p\n",
                               i, d->data[i]);
            }
    }
    up(&dev->sem);
    return 0;
}
```

这里，我们最终解释我们的" iterator" 值，简单地是一个 scull_dev 结构指针.

现在已有了一个完整的 iterator 操作的集合，scull 必须包装起它们，并且连接它们到 /proc 中的一个文件. 第一步是填充一个 seq_operations 结构:

```
static struct seq_operations scull_seq_ops = {
 .start = scull_seq_start,
 .next = scull_seq_next,
 .stop = scull_seq_stop,
 .show = scull_seq_show
};
```

有那个结构在，我们必须创建一个内核理解的文件实现. 我们不使用前面描述过的 read_proc 方法；在使用 seq_file 时，最好在一个稍低的级别上连接到 /proc. 那意味着创建一个 file_operations 结构(是的，和字符驱动使用的同样结构) 来实现所有内核需要的操作，来处理文件上的读和移动. 幸运的是，这个任务是简单的. 第一步是创建一个 open 方法连接文件到 seq_file 操作:

```
static int scull_proc_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &scull_seq_ops);
}
```

调用 seq_open 连接文件结构和我们上面定义的序列操作. 事实证明，open 是我们必须自己实现的唯一文件操作，因此我们现在可以建立我们的 file_operations 结构:

```
static struct file_operations scull_proc_ops = {
 .owner = THIS_MODULE,
 .open = scull_proc_open,
 .read = seq_read,
 .llseek = seq_lseek,
 .release = seq_release
};
```

这里我们指定我们自己的 open 方法，但是使用预装好的方法 seq_read, seq_lseek, 和 seq_release 给其他.

最后的步骤是创建 /proc 中的实际文件:

```
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

不是使用 create_proc_read_entry, 我们调用低层的 create_proc_entry, 我们有这个原型:

```
struct proc_dir_entry *create_proc_entry(const char *name,mode_t mode,struct proc_dir_entry *parent);
```

参数和它们的在 create_proc_read_entry 中的对等体相同: 文件名子, 它的位置, 以及父目录.

有了上面代码, scull 有一个新的 /proc 入口, 看来很象前面的一个. 但是, 它是高级的, 因为它不管它的输出有多么大, 它正确处理移动, 并且通常它是易读和易维护的. 我们建议使用 seq_file , 来实现包含多个非常小数目的输出行数的文件.

4.3.2. ioctl 方法

ioctl, 我们在第 1 章展示给你如何使用, 是一个系统调用, 作用于一个文件描述符; 它接收一个确定要进行的命令的数字和(可选地)另一个参数, 常常是一个指针. 作为一个使用 /proc 文件系统的替代, 你可以实现几个用来调试用的 ioctl 命令. 这些命令可以从驱动拷贝相关的数据结构到用户空间, 这里你可以检查它们.

这种方式使用 ioctl 来获取信息有些比使用 /proc 困难, 因为你需要另一个程序来发出 ioctl 并且显示结果. 必须编写这个程序, 编译, 并且与你在测试的模块保持同步. 另一方面, 驱动侧代码可能容易过需要实现一个 /proc 文件的代码.

有时候 ioctl 是获取信息最好的方法, 因为它运行比读取 /proc 快. 如果在数据写到屏幕之前必须做一些事情, 获取二进制形式的数据比读取一个文本文件要更有效. 另外, ioctl 不要求划分数据为小于一页的片段.

ioctl 方法的另一个有趣的优点是信息获取命令可留在驱动中, 当调试被禁止时. 不象对任何查看目录的人(并且太多人可能奇怪"这个怪文件是什么")都可见的 /proc 文件, 不记入文档的 ioctl 命令可能保持不为人知. 另外, 如果驱动发生了怪异的事情, 它们仍将在那里. 唯一的缺点是模块可能会稍微大些.

4.4. 使用观察来调试

有时小问题可以通过观察用户空间的应用程序的行为来追踪. 监视程序也有助于建立对驱动正确工作的信心. 例如, 我们能够对 scull 感到有信心, 在看了它的读实现如何响应不同数量数据的读请求之后.

有几个方法来监视用户空间程序运行. 你可以运行一个调试器来单步过它的函数, 增加打印语句, 或者在 strace 下运行程序. 这里, 我们将讨论最后一个技术, 当真正目的是检查内核代码时它是最有趣的.

strace 命令时一个有力工具, 显示所有的用户空间程序发出的系统调用. 它不仅显示调用, 还以符号形式显示调用的参数和返回值. 当一个系统调用失败, 错误的符号值(例如, ENOMEM)和对应的字串(Out of memory) 都显示. strace 有很多命令行选项; 其中最有用的是 -t 来显示每个调用执行的时间, -T 来显示调用中花费的时间, -e 来限制被跟踪调用的类型, 以及-o 来重定向输出到一个文件. 缺省地, strace 打印调用信息到 stderr.

strace 从内核自身获取信息. 这意味着可以跟踪一个程序, 不管它是否带有调试支持编译(对 gcc 是 -g 选项)以及不管它是否 strip 过. 你也可以连接追踪到一个运行中的进程, 类似于一个调试器的方式连接到一个运行中的进程并控制它.

跟踪信息常用来支持发给应用程序开发者的故障报告, 但是对内核程序员也是很有价值的. 我们已经看到驱动代码运行如何响应系统调用; strace 允许我们检查每个调用的输入和输出数据的一致性.

例如, 下面的屏幕输出显示(大部分)运行命令 strace ls /dev > /dev/scull0 的最后的行:

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0

fcntl64(3, F_SETFD, FD_CLOEXEC)  = 0
getdents64(3, /* 141 entries */, 4096)  = 4088
[...]
getdents64(3, /* 0 entries */, 4096)  = 0
close(3)  = 0
[...]

fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1) = 0
exit_group(0) = ?
```

从第一个 write 调用看, 明显地, 在 ls 结束查看目标目录后, 它试图写 4KB. 奇怪地(对 ls), 只有 4000 字节写入, 并且操作被重复. 但是, 我们知道 scull 中的写实现一次写一个单个量子, 因此我们本来就期望部分写. 几步之后, 所有东西清空, 程序成功退出.

作为另一个例子, 让我们读取 scull 设备(使用 wc 命令):

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865
read(3, "", 16384) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17) = 17
close(3) = 0
exit_group(0) = ?
```

如同期望的, read 一次只能获取 4000 字节, 但是数据总量等同于前个例子写入的. 注意在这个例子里读取是如何组织的, 同前面跟踪的相反. wc 为快速读被优化过, 因此绕过了标准库, 试图一个系统调用读取更多数据. 你可从跟踪的读的行里看到 wc 是如何试图一次读取 16 KB.

Linux 专家能够从 strace 的输出中发现更多有用信息. 如果你不想看到所有的符号, 你可使用 efile 标志来限制你自己仅查看文件方法是如何工作的.

就个人而言, 我们发现 strace 对于查明系统调用的运行时错误是非常有用. 常常是应用程序或演示程序中的 perror 调用不足够详细, 并且能够确切说出哪个系统调用的哪个参数触发了错误是非常有帮助的.

4.5. 调试系统故障

即便你已使用了所有的监视和调试技术, 有时故障还留在驱动里, 当驱动执行时系统出错. 当发生这个时, 能够收集尽可能多的信息来解决问题是重要的.

注意"故障"不意味着"崩溃". Linux 代码是足够健壮地优雅地响应大部分错误:一个故障常常导致当前进程的破坏而系统继续工作. 系统可能崩溃, 如果一个故障发生在一个进程的上下文之外, 或者如果系统的一些至关重要的部分毁坏了. 但是当是一个驱动错误导致的问题, 它常常只会导致不幸使用驱动的进程的突然死掉. 当进程被销毁时唯一无法恢复的破坏是分配给进程上下文的一些内存丢失了; 例如, 驱动通过 kmalloc 分配的动态列表可能丢失. 但是, 因为内核为任何一个打开的设备在进程死亡时调用关闭操作, 你的驱动可以释放由 open 方法分配的东西.

尽管一个 oops 常常都不会关闭整个系统, 你很有可能发现在发生一次后需要重启系统. 一个满是错误的驱动能使硬件处于不能使用的状态, 使内核资源处于不一致的状态, 或者, 最坏的情况, 在随机的地方破坏内核内存. 常常你可简单地卸载你的破驱动并且在一次 oops 后重试. 然而, 如果你看到任何东西建议说系统作为一个整体不太好了, 你最好立刻重启.

我们已经说过, 当内核代码出错, 一个提示性的消息打印在控制台上. 下一节解释如何解释并利用这样的消息. 尽管它们对新手看来相当模糊, 处理器转储是很有趣的信息, 常常足够来查明一个程序错误而不需要附加的测试.

4.5.1. oops 消息

大部分 bug 以解引用 NULL 指针或者使用其他不正确指针值来表现自己的. 此类 bug 通常的输出是一个 oops 消息.

处理器使用的任何地址几乎都是一个虚拟地址, 通过一个复杂的页表结构映射为物理地址(例外是内存管理子系统自己使用的物理地址). 当解引用一个无效的指针, 分页机制无法映射指针到一个物理地址, 处理器发出一个页错误给操作系统. 如果地址无效, 内核无法"页入"缺失的地址; 它(常常)产生一个 oops 如果在处理器处于管理模式时发生这个情况.

一个 oops 显示了出错时的处理器状态, 包括 CPU 寄存器内容和其他看来不可理解的信息. 消息由错误处理的 printk 语句产生(arch/*/kernel/traps.c)并且如同前面 "printk" 一节中描述的被分派.

我们看一个这样的消息. 这是来自在运行 2.6 内核的 PC 上一个 NULL 指针导致的结果. 这里最相关的信息是指令指针(EIP), 错误指令的地址.

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:  0
EIP:  0060:[<d083a064>]  Not tainted
EFLAGS: 00010246  (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000  ebx: 00000000  ecx: 00000000  edx: 00000000
esi: cf8b2460  edi: cf8b2480  ebp: 00000005  esp: c31c5f74
ds: 007b  es: 007b  ss: 0068

Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

写入一个由坏模块拥有的设备而产生的消息, 一个故意用来演示失效的模块. faulty.c 的 write 方法的实现是琐细的:

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t count,
loff_t *pos)
{
```

```
        /* make a simple fault by dereferencing a NULL pointer */
        *(int *)0 = 0;
        return 0;
}
```

如你能见, 我们这里做的是解引用一个 NULL 指针. 因为 0 一直是一个无效的指针值, 一个错误发生, 由内核转变为前面展示的 oops 消息. 调用进程接着被杀掉.

错误模块有不同的错误情况在它的读实现中:

```
ssize_t faulty_read(struct file *filp, char __user *buf, size_t count, loff_t
*pos)
{
    int ret;
    char stack_buf[4];

    /* Let's try a buffer overflow */
    memset(stack_buf, 0xff, 20);
    if (count > 4)

        count = 4; /* copy 4 bytes to the user */
    ret = copy_to_user(buf, stack_buf, count);
    if (!ret)

        return count;
    return ret;
}
```

这个方法拷贝一个字串到一个本地变量; 不幸的是, 字串长于目的数组. 当函数返回时导致的缓存区溢出引起一次 oops . 因为返回指令使指令指针到不知何处, 这类的错误很难跟踪, 并且你得到如下的:

```
EIP: 0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff

 printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU:  0
EIP:  0060:[<ffffffff>]  Not tainted
EFLAGS: 00010296  (2.6.6)
EIP is at 0xffffffff
eax: 0000000c  ebx: ffffffff  ecx: 00000000  edx: bfffda7c
esi: cf434f00  edi: ffffffff  ebp: 00002000  esp: c27fff78
ds: 007b  es: 007b  ss: 0068
```

```
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7
bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace: [<c0150612>] sys_read+0x42/0x70 [<c0103f8f>] syscall_call+0x7/0xb
Code: Bad EIP value.
```

这个情况，我们只看到部分的调用堆栈(vfs_read 和 faulty_read 丢失)，内核抱怨一个"坏 EIP 值". 这个抱怨和在开头列出的犯错的地址 (ffffffff) 都暗示内核堆栈已被破坏.

通常，当你面对一个 oops，第一件事是查看发生问题的位置，常常与调用堆栈分开列出. 在上面展示的第一个 oops，相关的行是:

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

这里我们看到，我们曾在函数 faulty_write，它位于 faulty 模块(在方括号中列出的). 16 进制数指示指令指针是函数内 4 字节，函数看来是 10 (16 进制)字节长. 常常这就足够来知道问题是什么.

如果你需要更多信息，调用堆栈展示给你如何得知在哪里坏事的. 堆栈自己是 16 机制形式打印的；做一点工作，你经常可以从堆栈的列表中决定本地变量的值和函数参数. 有经验的内核开发者可以从这里的某些模式识别中获益；例如，如果你看来自 faulty_read oops 的堆栈列表:

```
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7
 bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
 00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
```

堆栈顶部的 ffffffff 是我们坏事的字串的一部分. 在 x86 体系，缺省地，用户空间堆栈开始于 0xc0000000；因此，循环值 0xbfffda70 可能是一个用户堆栈地址；实际上，它是传递给 read 系统调用的缓存地址，每次下传过系统调用链时都被复制. 在 x86 (又一次，缺省地)，内核空间开始于 0xc0000000，因此这个之上的值几乎肯定是内核空间的地址，等等.

最后，当看一个 oops 列表，一直监视本章开始讨论的"slab 毒害"值. 例如,如果你得到一个内核 oops，里面的犯错地址时 0xa5a5a5a5，你几乎肯定 - 某个地方在初始化动态内存.

请注意，只在你的内核是打开 CONFIG_KALLSYMS 选项而编译时可以看到符号的调用堆栈. 否则，你见到一个裸的，16 机制列表，除非你以别的方式对其解码，它是远远无用的.

4.5.2. 系统挂起

尽管内核代码的大部分 bug 以 oops 消息结束，有时候它们可能完全挂起系统. 如果系统挂起，没有消息打印. 例如，如果代码进入一个无限循环，内核停止调度,[15]15 并且系统不会响应任何动作，包括魔术 Ctrl-Alt-Del 组合键. 你有 2 个选择来处理系统挂起-- 或者事先阻止它们，或者能够事后调试它们.

你可阻止无限循环通过插入 schedule 引用在战略点上. schedule 调用(如你可能猜到的)调度器，因此，允许别的进程从当前进程偷取 CPU 数据. 如果一个进程由于你的驱动的 bug 而在内核空间循环，schedule 调用使你能够杀掉进程在跟踪发生了什么之后.

你应当知道，当然，如何对 schedule 的调用可能创造一个附加的重入调用源到你的驱动，因为它允许别的进程运行. 这个重入正常地不应当是问题，假定你在你的驱动中已经使用了合适的加锁. 然而，要确认在你的驱动持有一个自旋锁的任何时间不能调用 schedule.

如果你的驱动真正挂起了系统，并且你不知道在哪里插入 schedule 调用，最好的方式是加入一些打印消息并且写到控制台(如果需要，改变 console_loglevel 值).

有时候系统可能看来被挂起，但是没有. 例如，这可能发生在键盘以某个奇怪的方式保持锁住的时候. 这些假挂起可通过查看你为此目的运行的程序的输出来检测. 一个你的显示器上的时钟或者系统负载表是一个好的状态监控器；只要他继续更新，调度器就在工作.

对许多的上锁一个必不可少的工具是"魔术 sysrq 键"，在大部分体系上都可用. 魔键 sysrq 是 PC 键盘上 alt 和 sysrq 键组合来发出的，或者在别的平台上使用其他特殊键(详见 documentation/sysrq.txt)，在串口控制台上也可用. 一个第三键，与这 2 个一起按下，进行许多有用的动作中的一个:

r 关闭键盘原始模式；用在一个崩溃的应用程序(例如 X 服务器)可能将你的键盘搞成一个奇怪的状态.

k 调用"安全注意键"(SAK) 功能. SAK 杀掉在当前控制台的所有运行的进程，给你一个干净的终端.

s 进行一个全部磁盘的紧急同步.

u umount. 试图重新加载所有磁盘在只读模式. 这个操作，常常在 s 之后马上调用，可以节省大量的文件系统检查时间，在系统处于严重麻烦时.

b boot. 立刻重启系统. 确认先同步和重新加载磁盘.

p 打印处理器消息.

t 打印当前任务列表.

m 打印内存信息.

---

15 [15] 实际上，多处理器系统仍然在其他处理器上调度，甚至一个单处理器的机器可能重新调度，如果内核抢占被使能. 然而，对于大部分的通常的情况(单处理器不使能抢占)，系统一起停止调度.

有别的魔术 sysrq 功能存在; 完整内容看内核源码的文档目录中的 sysrq.txt. 注意魔术 sysrq 必须在内核配置中显式使能, 大部分的发布没有使能它, 因为明显的安全理由. 对于用来开发驱动的系统, 然而, 使能魔术 sysrq 值得为它自己建立一个新内核的麻烦. 魔术 sysrq 可能在运行时关闭, 使用如下的一个命令:

```
echo 0 > /proc/sys/kernel/sysrq
```

如果非特权用户能够接触你的系统键盘, 你应当考虑关闭它, 来阻止有意或无意的损坏. 一些以前的内核版本缺省关闭 sysrq, 因此你需要在运行时使能它, 通过向同样的 /proc/sys 文件写入 1.

sysrq 操作是非常有用, 因此它们已经对不能接触到控制台的系统管理员可用. 文件 /proc/sysrq-trigger 是一个只写的入口点, 这里你可以触发一个特殊的 sysrq 动作, 通过写入关联的命令字符; 接着你可收集内核日志的任何输出数据. 这个 sysrq 的入口点是一直工作的, 即便 sysrq 在控制台上被关闭.

如果你经历一个"活挂", 就是你的驱动粘在一个循环中, 但是系统作为一个整体功能正常, 有几个技术值得了解. 经常地, sysrq p 功能直接指向出错的函数. 如果这个不行, 你还可以使用内核剖析功能. 建立一个打开剖析的内核, 并且用命令行中 profile=2 来启动它. 使用 readprofile 工具复位剖析计数器, 接着使你的驱动进入它的循环. 一会儿后, 使用 readprofile 来看内核在哪里消耗它的时间. 另一个更高级的选择是 oprofile, 你可以也考虑下. 文件 documentation/basic_profiling.txt 告诉你启动剖析器所有需要知道的东西.

在追逐系统挂起时一个值得使用的防范措施是以只读方式加载你的磁盘(或者卸载它们). 如果磁盘是只读或者卸载的, 就没有风险损坏文件系统或者使它处于不一致的状态. 另外的可能性是使用一个通过 NFS, 网络文件系统, 来加载它的全部文件系统的计算机, 内核的"NFS-Root"功能必须打开, 在启动时必须传递特殊的参数. 在这个情况下, 即便不依靠 sysrq 你也会避免文件系统破坏, 因为文件系统的一致有 NFS 服务器来管理, 你的设备驱动不会关闭它.

4.6. 调试器和相关工具

调试模块的最后手段是使用调试器来单步调试代码, 查看变量值和机器寄存器. 这个方法费时, 应当尽量避免. 但是, 通过调试器获得的代码的细粒度视角有时是很有价值的.

在内核上使用一个交互式调试器是一个挑战. 内核代表系统中的所有进程运行在自己的地址空间. 结果, 用户空间调试器所提供的一些普通功能, 例如断点和单步, 在内核中更难得到. 本节中, 我们看一下几个调试内核的方法; 每个都有缺点和优点.

4.6.1. 使用 gdb

gdb 对于看系统内部是非常有用. 在这个级别精通调试器的使用要求对 gdb 命令有信心, 需要理解目标平台的汇编代码, 以及对应源码和优化的汇编码的能力.

调试器必须把内核作为一个应用程序来调用. 除了指定内核映象的文件名之外, 你需要在命令行提供一个核心文件的名子. 对于一个运行的内核, 核心文件是内核核心映象, /proc/kcore. 一个典型的 gdb 调用看来如下:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是非压缩的 ELF 内核可执行文件的名子, 不是 zImage 或者 bzImage 或者给启动环境特别编译的任何东东.

gdb 命令行的第二个参数是核心文件的名子. 如同任何 /proc 中的文件, /proc/kcore 是在被读的时候产生的. 当 read 系统调用在 /proc 文件系统中执行时, 它映射到一个数据产生函数,而不是一个数据获取函数; 我们已经在本章"使用 /proc 文件系统"一节中利用了这个特点. kcore 用来代表内核"可执行文件", 以一个核心文件的形式; 它是一个巨大的文件, 因为他代表整个的内核地址空间, 对应于所有的物理内存. 从 gdb 中, 你可查看内核变量,通过发出标准 gdb 命令. 例如, p jiffies 打印时钟的从启动到当前时间的嘀哒数.

当你从 gdb 打印数据, 内核仍然在运行, 各种数据项在不同时间有不同的值; 然而, gdb 通过缓存已经读取的数据来优化对核心文件的存取. 如果你试图再次查看 jiffies 变量, 你会得到和以前相同的答案. 缓存值来避免额外的磁盘存取对传统核心文件是正确的做法, 但是在使用一个"动态"核心映象时就不方便. 解决方法是任何时候你需要刷新 gdb 缓存时发出命令 core-file /proc/kcore; 调试器准备好使用新的核心文件并且丢弃任何旧信息. 然而, 你不会一直需要发出 core-file 在读取一个新数据时; gdb 读取核心以多个几 KB 的块的方式, 并且只缓存它已经引用的块.

gdb 通常提供的不少功能在你使用内核时不可用. 例如, gdb 不能修改内核数据; 它希望在操作内存前在它自己的控制下运行一个被调试的程序. 也不可能设置断点或观察点, 或者单步过内核函数.

注意, 为了给 gdb 符号信息, 你必须设置 CONFIG_DEBUG_INFO 来编译你的内核. 结果是一个很大的内核映象在磁盘上, 但是, 没有这个信息, 深入内核变量几乎不可能.

有了调试信息, 你可以知道很多内核内部的事情. gdb 愉快地打印出结构, 跟随指针, 等等. 而有一个事情比较难, 然而, 是检查 modules. 因为模块不是传递给 gdb 的 vmlinux 映象, 调试器对它们一无所知. 幸运的是, 作为 2.6.7 内核, 有可能教给 gdb 需要如何检查可加载模块.

Linux 可加载模块是 ELF 格式的可执行映象; 这样, 它们被分成几个节. 一个典型的模块可能包含一打或更多节, 但是有 3 个典型的与一次调试会话相关:

.text

> 这个节包含有模块的可执行代码. 调试器必须知道在哪里以便能够给出回溯或者设置断点.(这些操作都不相关, 当运行一个调试器在 /proc/kcore 上, 但是它们在使用 kgdb 时可能有用, 下面描述).

.bss

.data

这 2 个节持有模块的变量. 在编译时不初始化的任何变量在 .bss 中, 而那些要初始化的在 .data 里.

使 gdb 能够处理可加载模块需要通知调试器一个给定模块的节加载在哪里. 这个信息在 sysfs 中, 在 /sys/module 下. 例如, 在加载 scull 模块后, 目录 /sys/module/scull/sections 包含名子为 .text 的文件; 每个文件的内容是那个节的基地址.

我们现在该发出一个 gdb 命令来告诉它关于我们的模块. 我们需要的命令是 add-symble-flile; 这个命令使用模块目标文件名, .text 基地址作为参数, 以及一系列描述任何其他感兴趣的节安放在哪里的参数. 在深入位于 sysfs 的模块节数据后, 我们可以构建这样一个命令:

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
 -s .data 0xd0836be0
```

我们已经包含了一个小脚本在例子代码里(gdbline), 它为给定的模块可以创建这个命令.

我们现在使用 gdb 检查我们的可加载模块中的变量. 这是一个取自 scull 调试会话的快速例子:

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
 -s .data 0xd0836be0
add symbol table from file "scull.ko" at
 .text_addr = 0xd0832000
 .bss_addr = 0xd0837100
 .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
 quantum = 4000,
 qset = 1000,
 size = 20881,
 access_key = 0,
 ...}
```

这里我们看到第一个 scull 设备当前持有 20881 字节. 如果我们想, 我们可以跟随数据链, 或者查看其他任何感兴趣的模块中的东东.

这是另一个值得知道的有用技巧:

```
(gdb) print *(address)
```

这里，填充 address 指向的一个 16 进制地址；输出是对应那个地址的代码的文件和行号. 这个技术可能有用，例如，来找出一个函数指针真正指向哪里.

我们仍然不能进行典型的调试任务，如设置断点或者修改数据；为进行这些操作，我们需要使用象 kdb(下面描述) 或者 kgdb (我们马上就到)这样的工具.

4.6.2. kdb 内核调试器

许多读者可能奇怪为什么内核没有建立更多高级的调试特性在里面.答案，非常简单，是 Linus 不相信交互式的调试器. 他担心它们会导致不好的修改，这些修改给问题打了补丁而不是找到问题的真正原因. 因此，没有内嵌的调试器.

其他内核开发者，但是，见到了交互式调试工具的一个临时使用. 一个这样的工具是 kdb 内嵌式内核调试器，作为来自 oss.sgi.com 的一个非官方补丁. 要使用 kdb，你必须获得这个补丁(确认获得一个匹配你的内核版本的版本)，应用它，重建并重安装内核. 注意，直到本书编写时，kdb 只在 IA-32(x86)系统中运行(尽管一个给 IA-64 的版本在主线内核版本存在了一阵子，在被去除之前.)

一旦你运行一个使能了 kdb 的内核，有几个方法进入调试器. 在控制台上按下 Pause(或者 Break) 键启动调试器. kdb 在一个内核 oops 发生时或者命中一个断点时也启动，在任何一种情况下，你看到象这样的一个消息：

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

注意，在 kdb 运行时内核停止任何东西. 在你调用 kdb 的系统中不应当运行其他东西；特别，你不应当打开网络 -- 除非，当然，你在调试一个网络驱动. 一般地以单用户模式启动系统是一个好主意，如果你将使用 kdb.

作为一个例子，考虑一个快速 scull 调试会话. 假设驱动已经加载，我们可以这样告诉 kdb 在 sucll_read 中设置一个断点：

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
 is enabled globally adjust 1
[0]kdb> go
```

bp 命令告诉 kdb 在下一次内核进入 scull_read 时停止. 你接着键入 go 来继续执行. 在将一些东西放入一个 scull 设备后，我们可以试着通过在另一个终端的外壳下运行 cat 命令来读取它，产生下面：

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read: int3

Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
```

```
Breakpoint @ 0xd087c5dc
[0]kdb>
```

我们现在位于 scull_read 的开始. 为看到我们任何到那里的, 我们可以获得一个堆栈回溯:

```
[0]kdb> bt
 ESP EIP Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb 试图打印出调用回溯中每个函数的参数. 然而, 它被编译器的优化技巧搞糊涂了. 因此, 它无法打印 scull_read 的参数.

到时候查看一些数据了. mds 命令操作数据; 我们可以查询 schull_devices 指针的值, 使用这样一个命令:

```
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00 ....
```

这里我们要求一个(4 字节)字, 起始于 scull_devices 的位置; 答案告诉我们的设备数组在地址 0xd0880de8; 第一个设备结构自己在 0xcf36ac00. 为查看那个设备结构, 我们需要使用这个地址:

```
[0]kdb> mds cf36ac00
0xcf36ac00 ce137dbc ....
0xcf36ac04 00000fa0 ....
0xcf36ac08 000003e8 ....
0xcf36ac0c 0000009b ....
0xcf36ac10 00000000 ....
0xcf36ac14 00000001 ....
0xcf36ac18 00000000 ....
0xcf36ac1c 00000001 ....
```

这里的 8 行对应于 scull_dev 结构的开始部分. 因此, 我们看到第一个设备的内存位于 0xce137dbc, quantum 是 4000 (16 进制 fa0), 量子集大小是 1000 (16 进制 3e8), 当前有 155(16 进制 9b) 字节存于设备中.

kdb 也可以改变数据. 假想我们要截短一些数据从设备中:

```
[0]kdb> mm cf26ac0c 0x50
0xcf26ac0c = 0x50
```

在设备上一个后续的 cat 会返回比之前少的数据.

kdb 有不少其他功能, 包括单步(指令, 不是 C 源码的一行), 在数据存取上设置断点, 反汇编代码, 步入链表, 存取寄存器数据, 还有更多. 在你应用了 kdb 补丁后, 一个完整的手册页集能够在你的源码树的 documentation/kdb 下发现.

4.6.3. kgdb 补丁

目前为止我们看到的 2 个交互式调试方法(使用 gdb 于 /proc/kcore 和 kdb) 都缺乏应用程序开发者已经熟悉的那种环境. 如果有一个真正的内核调试器支持改变变量, 断点等特色, 不是很好?

确实, 有这样一个解决方案. 在本书编写时, 2 个分开的补丁在流通中, 它允许 gdb, 具备完全功能, 针对内核运行. 这 2 个补丁都称为 kgdb. 它们通过分开运行测试内核的系统和运行调试器的系统来工作; 这 2 个系统典型地是通过一个串口线连接起来. 因此, 开发者可以在稳定地桌面系统上运行 gdb, 而操作一个运行在专门测试的盒子中的内核. 这种方式建立 gdb 开始需要一些时间, 但是很快会得到回报,当一个难问题出现时.

这些补丁目前处于健壮的状态, 在某些点上可能被合并, 因此我们避免说太多, 除了它们在哪里以及它们的基本特色. 鼓励感兴趣的读者去看这些的当前状态.

第一个 kgdb 补丁当前在 -mm 内核树里 -- 补丁进入 2.6 主线的集结场. 补丁的这个版本支持 x86, SuperH, ia64, x86_64, 和 32 位 PPC 体系. 除了通过串口操作的常用模式, 这个版本的 kgdb 可以通过一个局域网通讯. 使能以太网模式并且使用 kgdboe参数指定发出调试命令的 IP 地址来启动内核. 在 Documentation/i386/kgdb 下的文档描述了如何建立.[16] [16]

作为一个选择, 你可使用位于 http://kgdb.sf.net 的 kgdb 补丁. 这个调试器的版本不支持网络通讯模式(尽管据说在开发中), 但是它确实有内嵌的使用可加载模块的支持. 它支持 x86, x86_64, PowerPC, 和 S/390 体系.

4.6.4. 用户模式 Linux 移植

用户模式 Linux (UML) 是一个有趣的概念. 它被构建为一个分开的 Linux 内核移植, 有它自己的 arch/um 子目录. 它不在一个新的硬件类型上运行, 但是; 相反, 它运行在一个由 Linux 系统调用接口实现的虚拟机上. 如此, UML 使用 Linux 内核来运行, 作为一个 Linux 系统上的独立的用户模式进程.

有一个作为用户进程运行的内核拷贝有几个优点. 因为它们运行在一个受限的虚拟的处理器上, 一个错误的内核不能破坏"真实的"系统. 可以在同一台盒子轻易的尝试不同的硬件和软件配置. 并且, 也许对内核开发者而言, 用户模式内核可容易地使用 gdb 和 其他调试器操作.

毕竟, 它只是一个进程. UML 显然有加快内核开发的潜力.

[16] [16] 确实是忽略了指出, 你应当使你的网络适配卡建立在内核中, 然而, 否则调试器在启动时找不到它会关掉它自己.

然而, UML 有个大的缺点,从驱动编写者的角度看: 用户模式内核无法存取主机系统的硬件. 因此, 虽然它对于调试大部分本书的例子驱动是有用的, UML 对于不得不处理真实硬件的驱动的调试还是没有用处.

看 http://user-mode-linux.sf.net/ 关于 UML 的更多信息.

4.6.5. Linux 追踪工具

Linux Trace Toolkit (LTT) 是一个内核补丁以及一套相关工具, 允许追踪内核中的事件. 这个追踪包括时间信息, 可以创建一个给定时间段内发生事情的合理的完整图像. 因此, 它不仅用来调试也可以追踪性能问题.

LTT, 同广泛的文档一起, 可以在 http://www.opersys.com/LTT 找到.

4.6.6. 动态探针

Dynamic Probes (DProbes) 是由 IBM 发行的(在 GPL 之下)为 IA-32 体系的 Linux 的调试工具. 它允许安放一个"探针"在几乎系统中任何地方, 用户空间和内核空间都可以. 探针由一些代码组成(有一个特殊的,面向堆栈的语言写成), 当控制命中给定的点时执行. 这个代码可以报告信息给用户空间, 改变寄存器, 或者做其他很多事情. DProbes 的有用特性是, 一旦这个能力建立到内核中, 探针可以在任何地方插入在一个运行中的系统中, 不用内核建立或者重启. DProbes 可以和 LTT 一起来插入一个新的跟踪事件在任意位置.

DProbes 工具可以从 IBM 的开放源码网站:http://oss.sof-ware.ibm.com 下载.

第 5 章 并发和竞争情况

迄今，我们未曾关心并发的问题 -- 就是说，当系统试图一次做多件事时发生的情况. 然而，并发的管理是操作系统编程的核心问题之一. 并发相关的错误是一些最易出现又最难发现的问题. 即便是专家级 Linux 内核程序员偶尔也会出现并发相关的错误.

早期的 Linux 内核，较少有并发的源头. 内核不支持对称多处理器(SMP)系统，并发执行的唯一原因是硬件中断服务. 那个方法提供了简单性，但是在有越来越多处理器的系统上注重性能并且坚持系统要快速响应事件的世界中它不再可行了. 为响应现代硬件和应用程序的要求，Linux 内核已经发展为很多事情在同时进行. 这个进步已经产生了很大的性能和可扩展性. 然而，它也很大地使内核编程任务复杂化. 设备启动程序员现在必须从一开始就将并发作为他们设计的要素，并且他们必须对内核提供的并发管理设施有很强的理解.

本章的目的是开始建立那种理解的过程. 为此目的，我们介绍一些设施来立刻应用到第 3 章的 scull 驱动. 展示的其他设施暂时还不使用. 但是首先，我们看一下我们的简单 scull 驱动可能哪里出问题并且如何避免这些潜在的问题.

5.1. scull 中的缺陷

让我们快速看一段 scull 内存管理代码. 在写逻辑的深处，scull 必须决定它请求的内存是否已经分配. 处理这个任务的代码是:

```
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}
```

假设有 2 个进程(我们会称它们为"A"和"B") 独立地试图写入同一个 schull 设备的相同偏移. 每个进程同时到达上面片段的第一行的 if 测试. 如果被测试的指针是 NULL，每个进程都会决定分配内存，并且每个都会复制结果指针给 dptr->datat[s_pos]. 因为 2 个进程都在赋值给同一个位置，显然只有一个赋值可以成功.

当然，发生的是第 2 个完成赋值的进程将"胜出". 如果进程 A 先赋值，它的赋值将被进程 B 覆盖. 在此，scull 将完全忘记 A 分配的内存；它只有指向 B 的内存的指针. A 所分配的指针，因此，将被丢掉并且不再返回给系统.

事情的这个顺序是一个竞争情况的演示. 竞争情况是对共享数据的无控制存取的结果. 当错误的存取模式发生了，产生了不希望的东西. 对于这里讨论的竞争情况，结果是内存泄漏. 这已经足够坏了，但是竞争情况常常导致系统崩溃和数据损坏. 程序员可能被诱惑而忽视竞争情况为相当低可能性的事件. 但是，在计算机世界，百万分之一的事件会每隔几秒发生，并且后果会是严重的.

很快我们将去掉 scull 的竞争情况，但是首先我们需要对并发做一个更普遍的回顾.

5.2. 并发和它的管理

在现代 Linux 系统，有非常多的并发源，并且因此而来的可能竞争情况. 多个用户空间进程在运行，它们可能以令人惊讶的方式组合存取你的代码. SMP 系统能够同时在不同处理器上执行你的代码. 内核代码是可抢占的; 你的驱动代码可能在任何时间失去处理器，代替它的进程可能也在你的驱动中运行. 设备中断是能够导致你的代码并发执行的异步事件. 内核也提供各种延迟代码执行的机制，例如 workqueue，tasklet，以及定时器，这些能够使你的代码在任何时间以一种与当前进程在做的事情无关的方式运行. 在现代的，热插拔的世界中，你的设备可能在你使用它们的时候轻易地消失.

避免竞争情况可能是一个令人害怕的工作. 在一个任何时候可能发生任何事的世界，驱动程序员如何避免产生绝对的混乱? 事实证明，大部分竞争情况可以避免，通过一些想法，内核并发控制原语，以及几个基本原则的应用. 我们会先从原则开始，接着进入如何使用它们的细节中

竞争情况来自对资源的共享存取的结果. 当 2 个执行的线路[17][17]有机会操作同一个数据结构(或者硬件资源)，混合的可能性就一直存在. 因此第一个经验法则是在你设计驱动时在任何可能的时候记住避免共享的资源. 如果没有并发存取，就没有竞争情况. 因此小心编写的内核代码应当有最小的共享. 这个想法的最明显应用是避免使用全局变量. 如果你将一个资源放在多个执行线路能够找到它的地方，应当有一个很强的理由这样做.

事实是，然而，这样的共享常常是需要的. 硬件资源是，由于它们的特性，共享的，软件资源也必须常常共享给多个线程. 也要记住全局变量远远不是共享数据的唯一方式; 任何时候你的代码传递一个指针给内核的其他部分，潜在地它创造了一个新的共享情形. 共享是生活的事实.

这是资源共享的硬规则: 任何时候一个硬件或软件资源被超出一个单个执行线程共享，并且可能存在一个线程看到那个资源的不一致时，你必须明确地管理对那个资源的存取. 在上面的 scull 例子，这个情况在进程 B 看来是不一致的; 不知道进程 A 已经为(共享的) 设备分配了内存，它做它自己的分配并且覆盖了 A 的工作. 在这个例子里，我们必须控制对 scull 数据结构的存取. 我们需要安排，这样代码或者看到内存已经分配了，或者知道没有内存已经或者将要被其他人分配. 存取管理的常用技术是加锁或者互斥 -- 确保在任何时间只有一个执行线程可以操作一个共享资源. 本章剩下的大部分将专门介绍加锁.

然而，首先，我们必须简短考虑一下另一个重要规则. 当内核代码创建一个会被内核其他部分共享的对象时，这个对象必须一直存在(并且功能正常)到它知道没有对它的外部引用存在为止. scull 使它的设备可用的瞬间，它必须准备好处理对那些设备的请求. 并且 scull 必须一直能够处理对它的设备的请求直到它知道没有对这些设备的引用(例如打开的用户空间文件)存在. 2 个要求出自这个规则: 除非它处于可以正确工作的状态，不能有对象能对内核可用，对这样的对象的引用必须被跟踪. 在大部分情况下，你将发现内核为你处理引用计数，但是常常有例外.

[17] [17] 本章的意图，一个执行"线程"是任何运行代码的上下文. 每个进程显然是一个执行线程，但是一个中断处理也是，或者其他响应一个异步内核事件的代码.

遵照上面的规则需要计划和对细节小心注意. 容易被对资源的并发存取而吃惊, 你事先并没有认识到被共享. 通过一些努力, 然而, 大部分竞争情况能够在它们咬到你或者你的用户前被消灭.

5.3. 旗标和互斥体

让我们看看我们如何给 scull 加锁. 我们的目标是使我们对 scull 数据结构的操作原子化, 就是在有其他执行线程的情况下这个操作一次发生. 对于我们的内存泄漏例子, 我们需要保证, 如果一个线程发现必须分配一个特殊的内存块, 它有机会进行这个分配在其他线程可做测试之前. 为此, 我们必须建立临界区: 在任何给定时间只有一个线程可以执行的代码.

不是所有的临界区是同样的, 因此内核提供了不同的原语适用不同的需求. 在这个例子中, 每个对 scull 数据结构的存取都发生在由一个直接用户请求所产生的进程上下文中; 没有从中断处理或者其他异步上下文中的存取. 没有特别的周期(响应时间)要求; 应用程序程序员理解 I/O 请求常常不是马上就满足的. 进一步讲, scull 没有持有任何其他关键系统资源, 在它存取它自己的数据结构时. 所有这些意味着如果 scull 驱动在等待轮到它存取数据结构时进入睡眠, 没人介意.

"去睡眠" 在这个上下文中是一个明确定义的术语. 当一个 Linux 进程到了一个它无法做进一步处理的地方时, 它去睡眠(或者 "阻塞"), 让出处理器给别人直到以后某个时间它能够再做事情. 进程常常在等待 I/O 完成时睡眠. 随着我们深入内核, 我们会遇到很多情况我们不能睡眠. 然而 scull 中的 write 方法不是其中一个情况. 因此我们可使用一个加锁机制使进程在等待存取临界区时睡眠.

正如重要地, 我们将进行一个可能会睡眠的操作(使用 kmalloc 分配内存) -- 因此睡眠是一个在任何情况下的可能性. 如果我们的临界区要正确工作, 我们必须使用一个加锁原语在一个拥有锁的进程睡眠时起作用. 不是所有的加锁机制都能够在可能睡眠的地方使用(我们在本章后面会看到几个不可以的). 然而, 对我们现在的需要, 最适合的机制时一个旗标.

旗标在计算机科学中是一个被很好理解的概念. 在它的核心, 一个旗标是一个单个整型值, 结合有一对函数, 典型地称为 P 和 V. 一个想进入临界区的进程将在相关旗标上调用 P; 如果旗标的值大于零, 这个值递减 1 并且进程继续. 相反, 如果旗标的值是 0 (或更小), 进程必须等待直到别人释放旗标. 解锁一个旗标通过调用 V 完成; 这个函数递增旗标的值, 并且, 如果需要, 唤醒等待的进程.

当旗标用作互斥 -- 阻止多个进程同时在同一个临界区内运行 -- 它们的值将初始化为 1. 这样的旗标在任何给定时间只能由一个单个进程或者线程持有. 以这种模式使用的旗标有时称为一个互斥锁, 就是, 当然, "互斥"的缩写. 几乎所有在 Linux 内核中发现的旗标都是用作互斥.

5.3.1. Linux 旗标实现

Linux 内核提供了一个遵守上面语义的旗标实现, 尽管术语有些不同. 为使用旗标, 内核代码必须包含 <asm/semaphore.h>. 相关的类型是 struct semaphore; 实际旗标可以用几种方法来声明和初始化. 一种是直接创建一个旗标, 接着使用 sema_init 来设定它:

```
void sema_init(struct semaphore *sem, int val);
```

这里 val 是安排给旗标的初始值.

然而, 通常旗标以互斥锁的模式使用. 为使这个通用的例子更容易些, 内核提供了一套帮助函数和宏定义. 因此, 一个互斥锁可以声明和初始化, 使用下面的一种:

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

这里, 结果是一个旗标变量(称为 name), 初始化为 1 (使用 DECLARE_MUTEX) 或者 0 (使用 DECLARE_MUTEX_LOCKED). 在后一种情况, 互斥锁开始于上锁的状态; 在允许任何线程存取之前将不得不显式解锁它.

如果互斥锁必须在运行时间初始化(这是如果动态分配它的情况, 举例来说), 使用下列中的一个:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

在 Linux 世界中, P 函数称为 down -- 或者这个名子的某个变体. 这里, "down" 指的是这样的事实, 这个函数递减旗标的值, 并且, 也许在使调用者睡眠一会儿来等待旗标变可用之后, 给予对被保护资源的存取. 有 3 个版本的 down:

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

down 递减旗标值并且等待需要的时间. down_interruptible 同样, 但是操作是可中断的. 这个可中断的版本几乎一直是你要的那个; 它允许一个在等待一个旗标的用户空间进程被用户中断. 作为一个通用的规则, 你不想使用不可中断的操作, 除非实在是没有选择. 不可中断操作是一个创建不可杀死的进程(在 ps 中见到的可怕的 "D 状态")和惹恼你的用户的好方法, 使用 down_interruptible 需要一些格外的小心, 但是, 如果操作是可中断的, 函数返回一个非零值, 并且调用者不持有旗标. 正确的使用 down_interruptible 需要一直检查返回值并且针对性地响应.

最后的版本 (down_trylock) 从不睡眠; 如果旗标在调用时不可用, down_trylock 立刻返回一个非零值.

一旦一个线程已经成功调用 down 各个版本中的一个, 就说它持有着旗标(或者已经"取得"或者"获得"旗标). 这个线程现在有权力存取这个旗标保护的临界区. 当这个需要互斥的操作完成时, 旗标必须被返回. V 的 Linux 对应物是 up:

```
void up(struct semaphore *sem);
```

一旦 up 被调用, 调用者就不再拥有旗标.

如你所愿，要求获取一个旗标的任何线程，使用一个(且只能一个)对 up 的调用释放它. 在错误路径中常常需要特别的小心；如果在持有一个旗标时遇到一个错误，旗标必须在返回错误状态给调用者之前释放旗标. 没有释放旗标是容易犯的一个错误；这个结果(进程挂在看来无关的地方)可能是难于重现和跟踪的.

5.3.2. 在 scull 中使用旗标

旗标机制给予 scull 一个工具，可以在存取 scull_dev 数据结构时用来避免竞争情况. 但是正确使用这个工具是我们的责任. 正确使用加锁原语的关键是严密地指定要保护哪个资源并且确认每个对这些资源的存取都使用了正确的加锁方法. 在我们的例子驱动中，感兴趣的所有东西都包含在 scull_dev 结构里面，因此它是我们的加锁体制的逻辑范围.

让我们在看看这个结构:

```
struct scull_dev {
    struct scull_qset *data; /* Pointer to first quantum set */
    int quantum; /* the current quantum size */
    int qset; /* the current array size */
    unsigned long size; /* amount of data stored here */
    unsigned int access_key; /* used by sculluid and scullpriv */
    struct semaphore sem; /* mutual exclusion semaphore */
    struct cdev cdev; /* Char device structure */
};
```

到结构的底部是一个称为 sem 的成员，当然，它是我们的旗标. 我们已经选择为每个虚拟 scull 设备使用单独的旗标. 使用一个单个的全局的旗标也可能会是同样正确. 通常各种 scull 设备不共享资源，然而，并且没有理由使一个进程等待，而另一个进程在使用不同 scull 设备. 不同设备使用单独的旗标允许并行进行对不同设备的操作，因此，提高了性能.

旗标在使用前必须初始化. scull 在加载时进行这个初始化，在这个循环中:

```
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

注意，旗标必须在 scull 设备对系统其他部分可用前初始化. 因此，init_MUTEX 在 scull_setup_cdev 前被调用. 以相反的次序进行这个操作可能产生一个竞争情况，旗标可能在它准备好之前被存取.

下一步，我们必须浏览代码，并且确认在没有持有旗标时没有对 scull_dev 数据结构的存取. 因此，例如，scull_write 以这个代码开始:

```
if (down_interruptible(&dev->sem))
```

```
    return -ERESTARTSYS;
```

注意对 down_interruptible 返回值的检查; 如果它返回非零, 操作被打断了. 在这个情况下通常要做的是返回 -ERESTARTSYS. 看到这个返回值后, 内核的高层要么从头重启这个调用要么返回这个错误给用户. 如果你返回 -ERESTARTSYS, 你必须首先恢复任何用户可见的已经做了的改变, 以保证当重试系统调用时正确的事情发生. 如果你不能以这个方式恢复, 你应当替之返回 -EINTR.

scull_write 必须释放旗标, 不管它是否能够成功进行它的其他任务. 如果事事都顺利, 执行落到这个函数的最后几行:

```
out:
 up(&dev->sem);
 return retval;
```

这个代码释放旗标并且返回任何需要的状态. 在 scull_write 中有几个地方可能会出错; 这些地方包括内存分配失败或者在试图从用户空间拷贝数据时出错. 在这些情况中, 代码进行了一个 goto out, 以确保进行正确的清理.

5.3.3. 读者/写者旗标

旗标为所有调用者进行互斥, 不管每个线程可能想做什么. 然而, 很多任务分为 2 种清楚的类型: 只需要读取被保护的数据结构的类型, 和必须做改变的类型. 允许多个并发读者常常是可能的, 只要没有人试图做任何改变. 这样做能够显著提高性能; 只读的任务可以并行进行它们的工作而不必等待其他读者退出临界区.

Linux 内核为这种情况提供一个特殊的旗标类型称为 rwsem (或者" reader/writer semaphore"). rwsem 在驱动中的使用相对较少, 但是有时它们有用.

使用 rwsem 的代码必须包含 <linux/rwsem.h>. 读者写者旗标 的相关数据类型是 struct rw_semaphore; 一个 rwsem 必须在运行时显式初始化:

```
void init_rwsem(struct rw_semaphore *sem);
```

一个新初始化的 rwsem 对出现的下一个任务(读者或者写者)是可用的. 对需要只读存取的代码的接口是:

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

对 down_read 的调用提供了对被保护资源的只读存取, 与其他读者可能地并发地存取. 注意 down_read 可能将调用进程置为不可中断的睡眠. down_read_trylock 如果读存取是不可用时不会等待; 如果被准予存取它返回非零, 否则是 0. 注意 down_read_trylock 的惯例不同于大部分的内核函数, 返回值 0 指示成功. 一个使用 down_read 获取的 rwsem 必须最终使用 up_read 释放.

读者的接口类似:

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

down_write, down_write_trylock, 和 up_write 全部就像它们的读者对应部分, 除了, 当然, 它们提供写存取. 如果你处于这样的情况, 需要一个写者锁来做一个快速改变, 接着一个长时间的只读存取, 你可以使用 downgrade_write 在一旦你已完成改变后允许其他读者进入.

一个 rwsem 允许一个读者或者不限数目的读者来持有旗标. 写者有优先权; 当一个写者试图进入临界区, 就不会允许读者进入直到所有的写者完成了它们的工作. 这个实现可能导致读者饥饿 -- 读者被长时间拒绝存取 -- 如果你有大量的写者来竞争旗标. 由于这个原因, rwsem 最好用在很少请求写的时候, 并且写者只占用短时间.

5.4. Completions 机制

内核编程的一个普通模式包括在当前线程之外初始化某个动作, 接着等待这个动作结束. 这个动作可能是创建一个新内核线程或者用户空间进程, 对一个存在着的进程的请求, 或者一些基于硬件的动作. 在这些情况中, 很有诱惑去使用一个旗标来同步 2 个任务, 使用这样的代码:

```
struct semaphore sem;
init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

外部任务可以接着调用 up(??sem), 在它的工作完成时.

事实证明, 这种情况旗标不是最好的工具. 正常使用中, 试图加锁一个旗标的代码发现旗标几乎在所有时间都可用; 如果对旗标有很多竞争, 性能会受损并且加锁方案需要重新审视. 因此旗标已经对"可用"情况做了很多的优化. 当用上面展示的方法来通知任务完成, 然而, 调用 down 的线程将几乎是一直不得不等待; 因此性能将受损. 旗标还可能易于处于一个(困难的) 竞争情况, 如果它们表明为自动变量以这种方式使用时. 在一些情况中, 旗标可能在调用 up 的进程用完它之前消失.

这些问题引起了在 2.4.7 内核中增加了 "completion" 接口. completion 是任务使用的一个轻量级机制: 允许一个线程告诉另一个线程工作已经完成. 为使用 completion, 你的代码必须包含 <linux/completion.h>. 一个 completion 可被创建, 使用:

```
DECLARE_COMPLETION(my_completion);
```

或者, 如果 completion 必须动态创建和初始化:

```
struct completion my_completion;     /* ... */
```

```
init_completion(&my_completion);
```

等待 completion 是一个简单事来调用:

```
void wait_for_completion(struct completion *c);
```

注意这个函数进行一个不可打断的等待. 如果你的代码调用 wait_for_completion 并且没有人完成这个任务, 结果会是一个不可杀死的进程.[18] 18

另一方面, 真正的 completion 事件可能通过调用下列之一来发出:

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

如果多于一个线程在等待同一个 completion 事件, 这 2 个函数做法不同. complete 只唤醒一个等待的线程, 而 complete_all 允许它们所有都继续. 在大部分情况下, 只有一个等待者, 这 2 个函数将产生一致的结果.

一个 completion 正常地是一个单发设备; 使用一次就放弃. 然而, 如果采取正确的措施重新使用 completion 结构是可能的. 如果没有使用 complete_all, 重新使用一个 completion 结构没有任何问题, 只要对于发出什么事件没有模糊. 如果你使用 complete_all, 然而, 你必须在重新使用前重新初始化 completion 结构. 宏定义:

```
INIT_COMPLETION(struct completion c);
```

可用来快速进行这个初始化.

作为如何使用 completion 的一个例子, 考虑 complete 模块, 它包含在例子源码里. 这个模块使用简单的语义定义一个设备: 任何试图从一个设备读的进程将等待(使用 wait_for_completion)直到其他进程向这个设备写. 实现这个行为的代码是:

```
DECLARE_COMPLETION(comp);
ssize_t complete_read (struct file *filp, char __user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",current->pid,
current->comm);
    wait_for_completion(&comp);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t complete_write (struct file *filp, const char __user *buf, size_t
count, loff_t *pos)
{
```

18 [18] 在本书编写时, 添加可中断版本的补丁已经流行但是还没有合并到主线中.

```
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n", current->pid, current->comm);
    complete(&comp);
    return count; /* succeed, to avoid retrial */
}
```

有多个进程同时从这个设备"读"是有可能的. 每个对设备的写将确切地使一个读操作完成, 但是没有办法知道会是哪个.

completion 机制的典型使用是在模块退出时与内核线程的终止一起. 在这个原型例子里, 一些驱动的内部工作是通过一个内核线程在一个 while(1) 循环中进行的. 当模块准备好被清理时, exit 函数告知线程退出并且等待结束. 为此目的, 内核包含一个特殊的函数给线程使用:

```
void complete_and_exit(struct completion *c, long retval);
```

5.5. 自旋锁

对于互斥, 旗标是一个有用的工具, 但是它们不是内核提供的唯一这样的工具. 相反, 大部分加锁是由一种称为自旋锁的机制来实现. 不象旗标, 自旋锁可用在不能睡眠的代码中, 例如中断处理. 当正确地使用了, 通常自旋锁提供了比旗标更高的性能. 然而, 它们确实带来对它们用法的一套不同的限制.

自旋锁概念上简单. 一个自旋锁是一个互斥设备, 只能有 2 个值:"上锁"和"解锁". 它常常实现为一个整数值中的一个单个位. 想获取一个特殊锁的代码测试相关的位. 如果锁是可用的, 这个"上锁"位被置位并且代码继续进入临界区. 相反, 如果这个锁已经被别人获得, 代码进入一个紧凑的循环中反复检查这个锁, 直到它变为可用. 这个循环就是自旋锁的"自旋"部分.

当然, 一个自旋锁的真实实现比上面描述的复杂一点. 这个"测试并置位"操作必须以原子方式进行, 以便只有一个线程能够获得锁, 就算如果有多个进程在任何给定时间自旋. 必须小心以避免在超线程处理器上死锁 -- 实现多个虚拟 CPU 以共享一个单个处理器核心和缓存的芯片. 因此实际的自旋锁实现在每个 Linux 支持的体系上都不同. 核心的概念在所有系统上相同, 然而, 当有对自旋锁的竞争, 等待的处理器在一个紧凑循环中执行并且不作有用的工作.

它们的特性上, 自旋锁是打算用在多处理器系统上, 尽管一个运行一个抢占式内核的单处理器工作站的行为如同 SMP, 如果只考虑到并发. 如果一个非抢占的单处理器系统进入一个锁上的自旋, 它将永远自旋; 没有其他的线程再能够获得 CPU 来释放这个锁. 因此, 自旋锁在没有打开抢占的单处理器系统上的操作被优化为什么不作, 除了改变 IRQ 屏蔽状态的那些. 由于抢占, 甚至如果你从不希望你的代码在一个 SMP 系统上运行, 你仍然需要实现正确的加锁.

5.5.1. 自旋锁 API 简介

自旋锁原语要求的包含文件是 <linux/spinlock.h>. 一个实际的锁有类型 spinlock_t. 象任何其他数据结构, 一个 自旋锁必须初始化. 这个初始化可以在编译时完成, 如下:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者在运行时使用:

```
void spin_lock_init(spinlock_t *lock);
```

在进入一个临界区前, 你的代码必须获得需要的 lock , 用:

```
void spin_lock(spinlock_t *lock);
```

注意所有的自旋锁等待是, 由于它们的特性, 不可中断的. 一旦你调用 spin_lock, 你将自旋直到锁变为可用.

为释放一个你已获得的锁, 传递它给:

```
void spin_unlock(spinlock_t *lock);
```

有很多其他的自旋锁函数, 我们将很快都看到. 但是没有一个背离上面列出的函数所展示的核心概念. 除了加锁和释放, 没有什么可对一个锁所作的. 但是, 有几个规则关于你必须如何使用自旋锁. 我们将用一点时间来看这些, 在进入完整的自旋锁接口之前.

5.5.2. 自旋锁和原子上下文

想象一会儿你的驱动请求一个自旋锁并且在它的临界区里做它的事情. 在中间某处, 你的驱动失去了处理器. 或许它已调用了一个函数(copy_from_user, 假设) 使进程进入睡眠. 或者, 也许, 内核抢占发威, 一个更高优先级的进程将你的代码推到一边. 你的代码现在持有一个锁, 在可见的将来的如何时间不会释放这个锁. 如果某个别的线程想获得同一个锁, 它会, 在最好的情况下, 等待(在处理器中自旋)很长时间. 最坏的情况, 系统可能完全死锁.

大部分读者会同意这个场景最好是避免. 因此, 应用到自旋锁的核心规则是任何代码必须, 在持有自旋锁时, 是原子性的. 它不能睡眠; 事实上, 它不能因为任何原因放弃处理器, 除了服务中断(并且有时即便此时也不行)

内核抢占的情况由自旋锁代码自己处理. 内核代码持有一个自旋锁的任何时间, 抢占在相关处理器上被禁止. 即便单处理器系统必须以这种方式禁止抢占以避免竞争情况. 这就是为什么需要正确的加锁, 即便你从不期望你的代码在多处理器机器上运行.

在持有一个锁时避免睡眠是更加困难; 很多内核函数可能睡眠, 并且这个行为不是都被明确记录了. 拷贝数据到或从用户空间是一个明显的例子: 请求的用户空间页可能需要在拷贝进行前从磁盘上换入, 这个操作显然需要一个睡眠. 必须分配内存的任何操作都可能睡眠. kmalloc 能够决定放弃处理器, 并且等待更多内存可用除非它被明确告知不这样做. 睡眠可能发生在令人惊讶的地方; 编写会在自旋锁下执行的代码需要注意你调用的每个函数.

这有另一个场景: 你的驱动在执行并且已经获取了一个锁来控制对它的设备的存取. 当持有这个锁时, 设备发出一个中断, 使得你的中断处理运行. 中断处理, 在存取设备之前, 必须获得锁. 在一个中断处理中获取一个自旋锁是一个要做的合法的事情; 这是自旋锁操

作不能睡眠的其中一个理由. 但是如果中断处理和起初获得锁的代码在同一个处理器上会发生什么? 当中断处理在自旋, 非中断代码不能运行来释放锁. 这个处理器将永远自旋.

避免这个陷阱需要在持有自旋锁时禁止中断(只在本地 CPU). 有各种自旋锁函数会为你禁止中断(我们将在下一节见到它们). 但是, 一个完整的中断讨论必须等到第 10 章了.

关于自旋锁使用的最后一个重要规则是自旋锁必须一直是尽可能短时间的持有. 你持有一个锁越长, 另一个进程可能不得不自旋等待你释放它的时间越长, 它不得不完全自旋的机会越大. 长时间持有锁也阻止了当前处理器调度, 意味着高优先级进程 -- 真正应当能获得 CPU 的 -- 可能不得不等待. 内核开发者尽了很大努力来减少内核反应时间(一个进程可能不得不等待调度的时间)在 2.5 开发系列. 一个写的很差的驱动会摧毁所有的进程, 仅仅通过持有一个锁太长时间. 为避免产生这类问题, 重视使你的锁持有时间短.

5.5.3. 自旋锁函数

我们已经看到 2 个函数, spin_lock 和 spin_unlock, 可以操作自旋锁. 有其他几个函数, 然而, 有类似的名子和用途. 我们现在会展示全套. 这个讨论将带我们到一个我们无法在几章内适当涵盖的地方; 自旋锁 API 的完整理解需要对中断处理和相关概念的理解.

实际上有 4 个函数可以加锁一个自旋锁:

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
```

我们已经看到自旋锁如何工作. spin_loc_irqsave 禁止中断(只在本地处理器)在获得自旋锁之前; 之前的中断状态保存在 flags 里. 如果你绝对确定在你的处理器上没有禁止中断的(或者, 换句话说, 你确信你应当在你释放你的自旋锁时打开中断), 你可以使用 spin_lock_irq 代替, 并且不必保持跟踪 flags. 最后, spin_lock_bh 在获取锁之前禁止软件中断, 但是硬件中断留作打开的.

如果你有一个可能被在(硬件或软件)中断上下文运行的代码获得的自旋锁, 你必须使用一种 spin_lock 形式来禁止中断. 其他做法可能死锁系统, 迟早. 如果你不在硬件中断处理里存取你的锁, 但是你通过软件中断(例如, 在一个 tasklet 运行的代码, 在第 7 章涉及的主题), 你可以使用 spin_lock_bh 来安全地避免死锁, 而仍然允许硬件中断被服务.

也有 4 个方法来释放一个自旋锁; 你用的那个必须对应你用来获取锁的函数.

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

每个 spin_unlock 变体恢复由对应的 spin_lock 函数锁做的工作. 传递给 spin_unlock_irqrestore 的 flags 参数必须是传递给 spin_lock_irqsave 的同一个变

量. 你必须也调用 spin_lock_irqsave 和 spin_unlock_irqrestore 在同一个函数里. 否则, 你的代码可能破坏某些体系.

还有一套非阻塞的自旋锁操作:

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

这些函数成功时返回非零(获得了锁), 否则 0. 没有"try"版本来禁止中断.

5.5.4. 读者/写者自旋锁

内核提供了一个自旋锁的读者/写者形式, 直接模仿我们在本章前面见到的读者/写者旗标. 这些锁允许任何数目的读者同时进入临界区, 但是写者必须是排他的存取. 读者写者锁有一个类型 rwlock_t, 在 <linux/spinlokc.h> 中定义. 它们可以以 2 种方式被声明和被初始化:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* Dynamic way */
```

可用函数的列表现在应当看来相当类似. 对于读者, 下列函数是可用的:

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

有趣地, 没有 read_trylock. 对于写存取的函数是类似的:

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

读者/写者锁能够饿坏读者, 就像 rwsem 一样. 这个行为很少是一个问题; 然而, 如果有足够的锁竞争来引起饥饿, 性能无论如何都不行.

5.6. 锁陷阱

多年使用锁的经验 -- 早于 Linux 的经验 -- 已经表明加锁可能是非常难于正确的. 管理并发是一个固有的技巧性的事情, 有很多出错的方式. 在这一节, 我们快速看一下可能出错的东西.

5.6.1. 模糊的规则

如同上面已经说过的, 一个正确的加锁机制需要清晰和明确的规则. 当你创建一个可以被并发存取的资源时, 你应当定义哪个锁将控制存取. 加锁应当真正在开始处进行; 事后更改会是难的事情. 开始时花费的时间常常在调试时获得回报.

当你编写你的代码, 你会毫无疑问遇到几个函数需要存取通过一个特定锁保护的结构. 在此, 你必须小心: 如果一个函数需要一个锁并且接着调用另一个函数也试图请求这个锁, 你的代码死锁. 不论旗标还是自旋锁都不允许一个持锁者第 2 次请求锁; 如果你试图这样做, 事情就简单地完了.

为使的加锁正确工作, 你不得不编写一些函数, 假定它们的调用者已经获取了相关的锁. 常常地, 只有你的内部的, 静态函数能够这样编写; 从外部调用的函数必须明确处理加锁. 当你编写内部函数对加锁做了假设, 方便自己(和其他使用你的代码的人)并且明确记录这些假设. 在几个月后可能很难回来并记起是否你需要持有一个锁来调用一个特殊函数.

在 sucll 的例子里, 采用的设计决定是要求所有的函数直接从系统调用里调用, 来请求应用到被存取的设备结构上的旗标. 所有的内部函数, 那些只是从其他 scull 函数里调用的, 可以因此假设旗标已经正确获得.

5.6.2. 加锁顺序规则

在有大量锁的系统中(并且内核在成为这样一个系统), 一次需要持有多于一个锁, 对代码是不寻常的. 如果某类计算必须使用 2 个不同的资源进行, 每个有它自己的锁, 常常没有选择只能获取 2 个锁.

获得多个锁可能是危险的, 然而. 如果你有 2 个锁, 称为 Lock1 和 Lock2, 代码需要同时都获取, 你有一个潜在的死锁. 仅仅想象一个线程锁住 Lock1 而另一个同时获得 Lock2. 接着每个线程试图得到它没有的那个. 2 个线程都会死锁.

这个问题的解决方法常常是简单的: 当多个锁必须获得时, 它们应当一直以同样顺序获得. 只要遵照这个惯例, 象上面描述的简单死锁能够避免. 然而, 遵照加锁顺序规则是做比说难. 非常少见这样的规则真正在任何地方被写下. 常常你能做的最好的是看看别的代码如何做的.

一些经验规则能帮上忙. 如果你必须获得一个对你的代码来说的本地锁(假如, 一个设备锁), 以及一个属于内核更中心部分的锁, 先获取你的. 如果你有一个旗标和自旋锁的组

合，你必须，当然，先获得旗标；调用 down (可能睡眠) 在持有一个自旋锁时是一个严重的错误. 但是最重要的，尽力避免需要多于一个锁的情况.

5.6.3. 细 -粗- 粒度加锁

第一个支持多处理器系统的 Linux 内核是 2.0；它只含有一个自旋锁. 这个大内核锁将整个内核变为一个大的临界区；在任何时候只有一个 CPU 能够执行内核代码. 这个锁足够好地解决了并发问题以允许内核开发者从事所有其他的开发 SMP 所包含的问题. 但是它不是扩充地很好. 甚至一个 2 个处理器的系统可能花费可观数量的时间只是等待这个大内核锁. 一个 4 个处理器的系统的性能甚至不接近 4 个独立的机器的性能.

因此，后续的内核发布已经包含了更细粒度的加锁. 在 2.2 中，一个自旋锁控制对块 I/O 子系统的存取；另一个为网络而工作，等等. 一个现代的内核能包含几千个锁，每个保护一个小的资源. 这种细粒度的加锁可能对伸缩性是好的；它允许每个处理器在它自己特定的任务上工作而不必竞争其他处理器使用的锁. 很少人忘记大内核锁.[19] [19]

但是，细粒度加锁带有开销. 在有几千个锁的内核中，很难知道你需要那个锁 -- 以及你应当以什么顺序获取它们 -- 来进行一个特定的操作. 记住加锁错误可能非常难发现；更多的锁提供了更多的机会使真正有害的加锁 bug 钻进内核中. 细粒度加锁能带来一定水平的复杂性，长期来，对内核的可维护性有一个大的，不利的效果.

在一个设备驱动中加锁常常是相对直接的；你可以用一个锁来涵盖你做的所有东西，或者你可以给你管理的每个设备创建一个锁. 作为一个通用的规则，你应当从相对粗的加锁开始，除非你有确实的理由相信竞争可能是一个问题. 忍住怂恿去过早地优化；真实地性能约束常常表现在想不到的地方.

如果你确实怀疑锁竞争在损坏性能，你可能发现 lockmeter 工具有用. 这个补丁(从 http://oss.sgi.com/projects/lockmeter/ 可得到) 装备内核来测量在锁等待花费的时间. 通过看这个报告，你能够很快知道是否锁竞争真的是问题.

5.7. 加锁的各种选择

Linux 内核提供了不少有力的加锁原语能够用来使内核避免被自己绊倒. 但是，如同我们已见到的，一个加锁机制的设计和实现不是没有缺陷. 常常对于旗标和自旋锁没有选择；它们可能是唯一的方法来正确地完成工作. 然而，有些情况，可以建立原子的存取而不用完整的加锁. 本节看一下做事情的其他方法.

5.7.1. 不加锁算法

有时，你可以重新打造你的算法来完全避免加锁的需要. 许多读者/写者情况 -- 如果只有一个写者 -- 常常能够在这个方式下工作. 如果写者小心使数据结构的视图，由读者所见的，是一直一致的，有可能创建一个不加锁的数据结构.

---

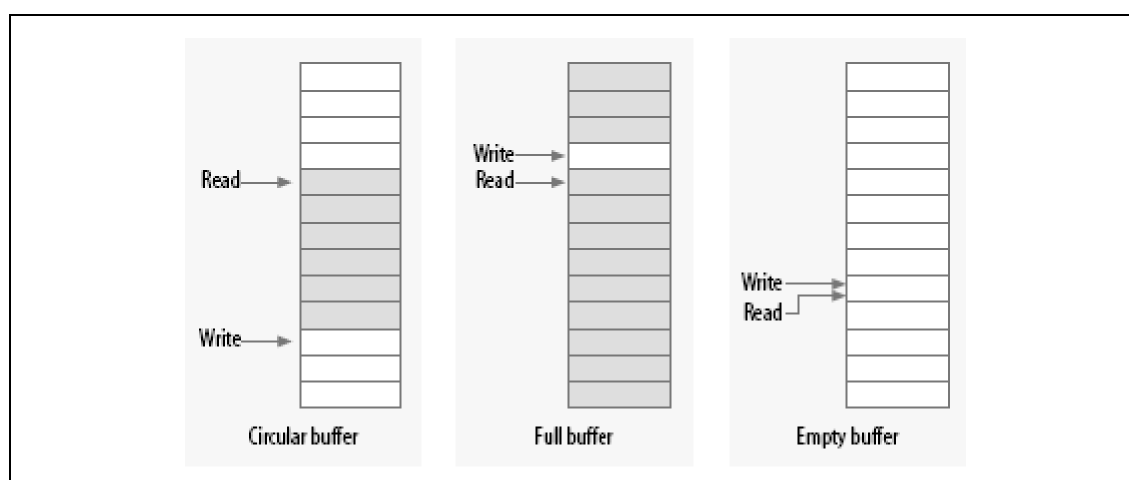
[19] [19] 这个锁仍然存在于 2.6，几个它现在覆盖内核非常小的部分. 如果你偶然发现一个 lock_kernel 调用，你已找到了这个大内核锁. 但是，想都不要想在任何新代码中使用它.

常常可以对无锁的生产者/消费者任务有用的数据结构是环形缓存. 这个算法包含一个生产者安放数据到一个数组的尾端, 而消费者从另一端移走数据. 当到达数组末端, 生产者绕回到开始. 因此一个环形缓存需要一个数组和 2 个索引值来跟踪下一个新值放到哪里, 以及哪个值在下一次应当从缓存中移走.

当小心地实现了, 一个环形缓存在没有多个生产者或消费者时不需要加锁. 生产者是唯一允许修改写索引和它所指向的数组位置的线程. 只要写者在更新写索引之前存储一个新值到缓存中, 读者将一直看到一个一致的视图. 读者, 轮换地, 是唯一存取读索引和它指向的值的线程. 加一点小心到确保 2 个指针不相互覆盖, 生产者和消费者可以并发存取缓存而没有竞争情况.

图环形缓冲展示了在几个填充状态的环形缓存. 这个缓存被定义成一个空情况由读写指针相同来指示, 而满情况发生在写指针紧跟在读指针后面的时候(小心解决绕回!). 当小心地编程, 这个缓存能够不必加锁地使用.

图 5.1. 环形缓冲



在设备驱动中环形缓存出现相当多. 网络适配器, 特别地, 常常使用环形缓存来与处理器交换数据(报文). 注意, 对于 2.6.10, 有一个通用的环形缓存实现在内核中可用; 如何使用它的信息看 <linux/kfifo.h>.

5.7.2. 原子变量

有时, 一个共享资源是一个简单的整数值. 假设你的驱动维护一个共享变量 n_op, 它告知有多少设备操作目前未完成. 正常地, 即便一个简单的操作例如:

```
n_op++;
```

可能需要加锁. 某些处理器可能以原子的方式进行那种递减, 但是你不能依赖它. 但是一个完整的加锁体制对于一个简单的整数值看来过分了. 对于这样的情况, 内核提供了一个原子整数类型称为 atomic_t, 定义在 <asm/atomic.h>.

一个 atomic_t 持有一个 int 值在所有支持的体系上. 但是, 因为这个类型在某些处理器上的工作方式, 整个整数范围可能不是都可用的; 因此, 你不应当指望一个 atomic_t

持有多于 24 位. 下面的操作为这个类型定义并且保证对于一个 SMP 计算机的所有处理器来说是原子的. 操作是非常快的, 因为它们在任何可能时编译成一条单个机器指令.

```
void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
```

设置原子变量 v 为整数值 i. 你也可在编译时使用宏定义 ATOMIC_INIT 初始化原子值.

```
int atomic_read(atomic_t *v);
```

返回 v 的当前值.

```
void atomic_add(int i, atomic_t *v);
```

由 v 指向的原子变量加 i. 返回值是 void, 因为有一个额外的开销来返回新值, 并且大部分时间不需要知道它.

```
void atomic_sub(int i, atomic_t *v);
```

从 *v 减去 i.

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

递增或递减一个原子变量.

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

进行一个特定的操作并且测试结果; 如果, 在操作后, 原子值是 0, 那么返回值是真; 否则, 它是假. 注意没有 atomic_add_and_test.

```
int atomic_add_negative(int i, atomic_t *v);
```

加整数变量 i 到 v. 如果结果是负值返回值是真, 否则为假.

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

就像 atomic_add 和其类似函数, 除了它们返回原子变量的新值给调用者.

如同它们说过的, atomic_t 数据项必须通过这些函数存取. 如果你传递一个原子项给一个期望一个整数参数的函数, 你会得到一个编译错误.

你还应当记住，atomic_t 值只在当被置疑的量真正是原子的时候才起作用. 需要多个 atomic_t 变量的操作仍然需要某种其他种类的加锁. 考虑一下下面的代码:

```
atomic_sub(amount, &first_atomic);
atomic_add(amount, &second_atomic);
```

从第一个原子值中减去 amount，但是还没有加到第二个时，存在一段时间. 如果事情的这个状态可能产生麻烦给可能在这 2 个操作之间运行的代码，某种加锁必须采用.

5.7.3. 位操作

atomic_t 类型在进行整数算术时是不错的. 但是，它无法工作的好，当你需要以原子方式操作单个位时. 为此，内核提供了一套函数来原子地修改或测试单个位. 因为整个操作在单步内发生，没有中断(或者其他处理器)能干扰.

原子位操作非常快，因为它们使用单个机器指令来进行操作，而在任何时候低层平台做的时候不用禁止中断. 函数是体系依赖的并且在 <asm/bitops.h> 中声明. 它们保证是原子的，即便在 SMP 计算机上，并且对于跨处理器保持一致是有用的.

不幸的是，键入这些函数中的数据也是体系依赖的. nr 参数(描述要操作哪个位)常常定义为 int，但是在几个体系中是 unsigned long. 要修改的地址常常是一个 unsigned long 指针，但是几个体系使用 void * 代替.

各种位操作是:

```
void set_bit(nr, void *addr);
```

设置第 nr 位在 addr 指向的数据项中.

```
void clear_bit(nr, void *addr);
```

清除指定位在 addr 处的无符号长型数据. 它的语义与 set_bit 的相反.

```
void change_bit(nr, void *addr);
```

翻转这个位.

```
test_bit(nr, void *addr);
```

这个函数是唯一一个不需要是原子的位操作; 它简单地返回这个位的当前值.

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

原子地动作如同前面列出的，除了它们还返回这个位以前的值.

当这些函数用来存取和修改一个共享的标志，除了调用它们不用做任何事；它们以原子发生进行它们的操作．使用位操作来管理一个控制存取一个共享变量的锁变量，另一方面，是有点复杂并且应该有个例子．大部分现代的代码不以这种方法来使用位操作，但是象下面的代码仍然在内核中存在．

一段需要存取一个共享数据项的代码试图原子地请求一个锁，使用 test_and_set_bit 或者 test_and_clear_bit．通常的实现展示在这里；它假定锁是在地址 addr 的 nr 位．它还假定当锁空闲是这个位是 0，忙为 非零．

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();

/* do your work */

/* release lock, and check... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong(); /* already released: error */
```

如果你通读内核源码，你会发现象这个例子的代码．但是，最好在新代码中使用自旋锁；自旋锁很好地调试过，它们处理问题如同中断和内核抢占，并且别人读你代码时不必努力理解你在做什么．

5.7.4. seqlock 锁

2.6 内核包含了一对新机制打算来提供快速地，无锁地存取一个共享资源．seqlock 在这种情况下工作，要保护的资源小，简单，并且常常被存取，并且很少写存取但是必须要快．基本上，它们通过允许读者释放对资源的存取，但是要求这些读者来检查与写者的冲突而工作，并且当发生这样的冲突时，重试它们的存取．seqlock 通常不能用在保护包含指针的数据结构，因为读者可能跟随着一个无效指针而写者在改变数据结构．

seqlock 定义在 <linux/seqlock.h>．有 2 个通常的方法来初始化一个 seqlock(有 seqlock_t 类型):

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

读存取通过在进入临界区入口获取一个(无符号的)整数序列来工作．在退出时，那个序列值与当前值比较；如果不匹配，读存取必须重试．结果是，读者代码象下面的形式:

```
unsigned int seq;

do {
    seq = read_seqbegin(&the_lock);
```

```
    /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

这个类型的锁常常用在保护某种简单计算，需要多个一致的值．如果这个计算最后的测试表明发生了一个并发的写，结果被简单地丢弃并且重新计算．

如果你的 seqlock 可能从一个中断处理里存取，你应当使用 IRQ 安全的版本来代替：

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long
flags);
```

写者必须获取一个排他锁来进入由一个 seqlock 保护的临界区．为此，调用：

```
void write_seqlock(seqlock_t *lock);
```

写锁由一个自旋锁实现，因此所有的通常的限制都适用．调用：

```
void write_sequnlock(seqlock_t *lock);
```

来释放锁．因为自旋锁用来控制写存取，所有通常的变体都可用：

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

还有一个 write_tryseqlock 在它能够获得锁时返回非零．

5.7.5. 读取-拷贝-更新

读取-拷贝-更新(RCU) 是一个高级的互斥方法，能够有高效率在合适的情况下．它在驱动中的使用很少但是不是没人知道，因此这里值得快速浏览下．那些感兴趣 RCU 算法的完整细节的人可以在由它的创建者出版的白皮书中找到(http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html).

RCU 对它所保护的数据结构设置了不少限制．它对经常读而极少写的情况做了优化．被保护的资源应当通过指针来存取，并且所有对这些资源的引用必须由原子代码持有．当数据结构需要改变，写线程做一个拷贝，改变这个拷贝，接着使相关的指针对准新的版本 -- 因此，有了算法的名子．当内核确认没有留下对旧版本的引用，它可以被释放．

作为在真实世界中使用 RCU 的例子，考虑一下网络路由表．每个外出的报文需要请求检查路由表来决定应当使用哪个接口．这个检查是快速的，并且，一旦内核发现了目标接口，它不再需要路由表入口项．RCU 允许路由查找在没有锁的情况下进行，具有相当多的性能好处．内核中的 Startmode 无线 IP 驱动也使用 RCU 来跟踪它的设备列表．

使用 RCU 的代码应当包含 <linux/rcupdate.h>.

在读这一边，使用一个 RCU-保护的数据结构的代码应当用 rcu_read_lock 和 rcu_read_unlock 调用将它的引用包含起来. 结果就是，RCU 代码往往是象这样:

```
struct my_stuff *stuff;
rcu_read_lock();
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock();
```

rcu_read_lock 调用是快的; 它禁止内核抢占但是没有等待任何东西. 在读"锁"被持有时执行的代码必须是原子的. 在对 rcu_read_unlock 调用后，没有使用对被保护的资源的引用.

需要改变被保护的结构的代码必须进行几个步骤. 第一步是容易的; 它分配一个新结构，如果需要就从旧的拷贝数据，接着替换读代码所看到的指针. 在此，对于读一边的目的，改变结束了. 任何进入临界区的代码看到数据的新版本.

剩下的是释放旧版本. 当然，问题是在其他处理器上运行的代码可能仍然有对旧数据的一个引用，因此它不能立刻释放. 相反，写代码必须等待直到它知道没有这样的引用存在了. 因为所有持有对这个数据结构引用的代码必须(规则规定)是原子的，我们知道一旦系统中的每个处理器已经被调度了至少一次，所有的引用必须消失. 这就是 RCU 所做的; 它留下了一个等待直到所有处理器已经调度的回调; 那个回调接下来被运行来进行清理工作.

改变一个 RCU-保护的数据结构的代码必须通过分配一个 struct rcu_head 来获得它的清理回调，尽管不需要以任何方式初始化这个结构. 常常，那个结构被简单地嵌入在 RCU 所保护的大的资源里面. 在改变资源完成后，应当调用:

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

给定的 func 在释放资源是安全的时候调用; 传递给 call_rcu 的是给同一个 arg. 常常，func 需要的唯一的东西是调用 kfree.

全部 RCU 接口比我们已见的要更加复杂; 它包括，例如，辅助函数来使用被保护的链表. 全部内容见相关的头文件.

5.8. 快速参考

本章已介绍了很多符号给并发的管理. 最重要的这些在此总结:

#include <asm/semaphore.h>

定义旗标和其上操作的包含文件.

DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);

2 个宏定义，用来声明和初始化一个在互斥模式下使用的旗标.

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

这 2 函数用来在运行时初始化一个旗标.

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

加锁和解锁旗标. down 使调用进程进入不可打断睡眠，如果需要; down_interruptible，相反，可以被信号打断. down_trylock 不睡眠; 相反，它立刻返回如果旗标不可用. 加锁旗标的代码必须最终使用 up 解锁它.

```
struct rw_semaphore;
init_rwsem(struct rw_semaphore *sem);
```

旗标的读者/写者版本和初始化它的函数.

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

获得和释放对读者/写者旗标的读存取的函数.

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

管理对读者/写者旗标写存取的函数.

```
#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c);
```

描述 Linux completion 机制的包含文件，已经初始化 completion 的正常方法. INIT_COMPLETION 应当只用来重新初始化一个之前已经使用过的 completion.

```
void wait_for_completion(struct completion *c);
```

等待一个 completion 事件发出.

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

发出一个 completion 事件. completion 唤醒, 最多, 一个等待着的线程, 而 complete_all 唤醒全部等待者.

```
void complete_and_exit(struct completion *c, long retval);
```

通过调用 complete 来发出一个 completion 事件, 并且为当前线程调用 exit.

```
#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
```

定义自旋锁接口的包含文件, 以及初始化锁的 2 个方法.

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```

加锁一个自旋锁的各种方法, 并且, 可能地, 禁止中断.

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

上面函数的非自旋版本; 在获取锁失败时返回 0, 否则非零.

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

释放一个自旋锁的相应方法.

```
rwlock_t lock = RW_LOCK_UNLOCKED
rwlock_init(rwlock_t *lock);
```

初始化读者/写者锁的 2 个方法.

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

获得一个读者/写者锁的读存取的函数.

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

释放一个读者/写者自旋锁的读存取.

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
```

获得一个读者/写者锁的写存取的函数.

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

释放一个读者/写者自旋锁的写存取的函数.

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

原子地存取整数变量. atomic_t 变量必须只通过这些函数存取.

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

原子地存取位值; 它们可用做标志或者锁变量. 使用这些函数阻止任何与并发存取这个位相关的竞争情况.

```
#include <linux/seqlock.h>
seqlock_t lock = SEQLOCK_UNLOCKED;
seqlock_init(seqlock_t *lock);
```

定义 seqlock 的包含文件, 已经初始化它们的 2 个方法.

```
unsigned int read_seqbegin(seqlock_t *lock);
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry(seqlock_t *lock, unsigned int seq);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long
flags);
```

获得一个 seqlock-保护 的资源的读权限的函数.

```
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
```

获取一个 seqlock-保护的资源的写权限的函数.

```
void write_sequnlock(seqlock_t *lock);
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

释放一个 seqlock-保护的资源的写权限的函数.

```
#include <linux/rcupdate.h>
```

需要使用读取-拷贝-更新(RCU)机制的包含文件.

```
void rcu_read_lock;
void rcu_read_unlock;
```

获取对由 RCU 保护的资源的原子读权限的宏定义.

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

安排一个回调在所有处理器已经被调度以及一个 RCU-保护的资源可用被安全的释放之后运行.

第 6 章 高级字符驱动操作

在第 3 章，我们建立了一个完整的设备驱动，用户可用来写入和读取．但是一个真正的设备常常提供比同步读和写更多的功能．现在我们已装备有调试工具如果发生错误，并且一个牢固的并发的理解来帮助避免事情进入错误-- 我们可安全地前进并且创建一个更高级的驱动.

本章检查几个你需要理解的概念来编写全特性的字符设备驱动．我们从实现 ioctl 系统调用开始，它是用作设备控制的普通接口．接着我们进入各种和用户空间同步的方法；在本章结尾，你有一个充分的认识对于如何使进程睡眠(并且唤醒它们)，实现非阻塞的 I/O，并且通知用户空间当你的设备可用来读或写．我们以查看如何在驱动中实现几个不同的设备存取策略来结束.

这里讨论的概念通过 scull 驱动的几个修改版本来演示．再一次，所有的都使用内存中的虚拟设备来实现，因此你可自己试验这些代码而不必使用任何特别的硬件．到此为止，你可能在想亲手使用真正的硬件，但是那将必须等到第 9 章.

6.1. ioctl 接口

大部分驱动需要 -- 除了读写设备的能力 -- 通过设备驱动进行各种硬件控制的能力．大部分设备可进行超出简单的数据传输之外的操作；用户空间必须常常能够请求，例如，设备锁上它的门，弹出它的介质，报告错误信息，改变波特率，或者自我销毁．这些操作常常通过 ioctl 方法来支持，它通过相同名子的系统调用来实现.

在用户空间，ioctl 系统调用有下面的原型:

```
int ioctl(int fd, unsigned long cmd, ...);
```

这个原型由于这些点而凸现于 Unix 系统调用列表，这些点常常表示函数有数目不定的参数．在实际系统中，但是，一个系统调用不能真正有变数目的参数．系统调用必须有一个很好定义的原型，因为用户程序可存取它们只能通过硬件的"门"．因此，原型中的点不表示一个变数目的参数，而是一个单个可选的参数，传统上标识为 char *argp．这些点在那里只是为了阻止在编译时的类型检查．第 3 个参数的实际特点依赖所发出的特定的控制命令(第 2 个参数)．一些命令不用参数，一些用一个整数值，以及一些使用指向其他数据的指针．使用一个指针是传递任意数据到 ioctl 调用的方法；设备接着可与用户空间交换任何数量的数据.

ioctl 调用的非结构化特性使它在内核开发者中失宠．每个 ioctl 命令，基本上，是一个单独的，常常无文档的系统调用，并且没有方法以任何类型的全面的方式核查这些调用．也难于使非结构化的 ioctl 参数在所有系统上一致工作；例如，考虑运行在 32-位模式的一个用户进程的 64-位 系统．结果，有很大的压力来实现混杂的控制操作，只通过任何其他的方法．可能的选择包括嵌入命令到数据流(本章稍后我们将讨论这个方法)或者使用虚拟文件系统，要么是 sysfs 要么是设备特定的文件系统．(我们将在 14 章看看 sysfs)．但是，事实是 ioctl 常常是最容易的和最直接的选择,对于真正的设备操作.

ioctl 驱动方法有和用户空间版本不同的原型:

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);
```

inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数. cmd 参数从用户那里不改变地传下来, 并且可选的参数 arg 参数以一个 unsigned long 的形式传递, 不管它是否由用户给定为一个整数或一个指针. 如果调用程序不传递第 3 个参数, 被驱动操作收到的 arg 值是无定义的. 因为类型检查在这个额外参数上被关闭, 编译器不能警告你如果一个无效的参数被传递给 ioctl, 并且任何关联的错误将难以查找.

如果你可能想到的, 大部分 ioctl 实现包括一个大的 switch 语句来根据 cmd 参数, 选择正确的做法. 不同的命令有不同的数值, 它们常常被给予符号名来简化编码. 符号名通过一个预处理定义来安排. 定制的驱动常常声明这样的符号在它们的头文件中; scull.h 为 scull 声明它们. 用户程序必须, 当然, 包含那个头文件来存取这些符号.

6.1.1. 选择 ioctl 命令

在为 ioctl 编写代码之前, 你需要选择对应命令的数字. 许多程序员的第一个本能的反应是选择一组小数从 0 或 1 开始, 并且从此开始向上. 但是, 有充分的理由不这样做. ioctl 命令数字应当在这个系统是唯一的, 为了阻止向错误的设备发出正确的命令而引起的错误. 这样的不匹配不会不可能发生, 并且一个程序可能发现它自己试图改变一个非串口输入系统的波特率, 例如一个 FIFO 或者一个音频设备. 如果这样的 ioctl 号是唯一的, 这个应用程序得到一个 EINVAL 错误而不是继续做不应当做的事情.

为帮助程序员创建唯一的 ioctl 命令代码, 这些编码已被划分为几个位段. Linux 的第一个版本使用 16-位数: 高 8 位是关联这个设备的"魔"数, 低 8 位是一个顺序号, 在设备内唯一. 这样做是因为 Linus 是"无能"的(他自己的话); 一个更好的位段划分仅在后来被设想. 不幸的是, 许多驱动仍然使用老传统. 它们不得不: 改变命令编码会破坏大量的二进制程序,并且这不是内核开发者愿意见到的.

根据 Linux 内核惯例来为你的驱动选择 ioctl 号, 你应当首先检查 include/asm/ioctl.h 和 Documentation/ioctl-number.txt. 这个头文件定义你将使用的位段: type(魔数), 序号, 传输方向, 和参数大小. ioctl-number.txt 文件列举了在内核中使用的魔数,[20][20] 因此你将可选择你自己的魔数并且避免交叠. 这个文本文件也列举了为什么应当使用惯例的原因.

定义 ioctl 命令号的正确方法使用 4 个位段, 它们有下列的含义. 这个列表中介绍的新符号定义在 <linux/ioctl.h>.

type

---

[20] [20] 但是, 这个文件的维护在后来有些少见了.

魔数. 只是选择一个数(在参考了 ioctl-number.txt 之后)并且使用它在整个驱动中. 这个成员是 8 位宽(_IOC_TYPEBITS).

number

序(顺序)号. 它是 8 位(_IOC_NRBITS)宽.

direction

数据传送的方向,如果这个特殊的命令涉及数据传送. 可能的值是 _IOC_NONE(没有数据传输), _IOC_READ, _IOC_WRITE, 和 _IOC_READ|_IOC_WRITE (数据在 2 个方向被传送). 数据传送是从应用程序的观点来看待的; _IOC_READ 意思是从设备读, 因此设备必须写到用户空间. 注意这个成员是一个位掩码, 因此 _IOC_READ 和 _IOC_WRITE 可使用一个逻辑 AND 操作来抽取.

size

涉及到的用户数据的大小. 这个成员的宽度是依赖体系的, 但是常常是 13 或者 14 位. 你可为你的特定体系在宏 _IOC_SIZEBITS 中找到它的值. 你使用这个 size 成员不是强制的 - 内核不检查它 -- 但是它是一个好主意. 正确使用这个成员可帮助检测用户空间程序的错误并使你实现向后兼容, 如果你曾需要改变相关数据项的大小. 如果你需要更大的数据结构, 但是, 你可忽略这个 size 成员. 我们很快见到如何使用这个成员.

头文件 <asm/ioctl.h>, 它包含在 <linux/ioctl.h> 中, 定义宏来帮助建立命令号, 如下: _IO(type,nr)(给没有参数的命令), _IOR(type, nre, datatype)(给从驱动中读数据的), _IOW(type,nr,datatype)(给写数据), 和 _IOWR(type,nr,datatype)(给双向传送). type 和 number 成员作为参数被传递, 并且 size 成员通过应用 sizeof 到 datatype 参数而得到.

这个头文件还定义宏, 可被用在你的驱动中来解码这个号: _IOC_DIR(nr), _IOC_TYPE(nr), _IOC_NR(nr), 和 _IOC_SIZE(nr). 我们不进入任何这些宏的细节, 因为头文件是清楚的, 并且在本节稍后有例子代码展示.

这里是一些 ioctl 命令如何在 scull 被定义的. 特别地, 这些命令设置和获得驱动的可配置参数.

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC 'k'
/* Please use a different 8-bit number in your code */

#define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)
/*
 * S means "Set" through a ptr,
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
```

```
 * X means "eXchange": switch G and S atomically
 * H means "sHift": switch T and Q atomically
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
#define SCULL_IOCSQSET _IOW(SCULL_IOC_MAGIC, 2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC, 3)
#define SCULL_IOCTQSET _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOCGQSET _IOR(SCULL_IOC_MAGIC, 6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 7)
#define SCULL_IOCQQSET _IO(SCULL_IOC_MAGIC, 8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQSET _IO(SCULL_IOC_MAGIC, 12)

#define SCULL_IOC_MAXNR 14
```

真正的源文件定义几个额外的这里没有出现的命令.

我们选择实现 2 种方法传递整数参数: 通过指针和通过明确的值(尽管, 由于一个已存在的惯例, ioclt 应当通过指针交换值). 类似地, 2 种方法被用来返回一个整数值:通过指针和通过设置返回值. 这个有效只要返回值是一个正的整数; 如同你现在所知道的, 在从任何系统调用返回时, 一个正值被保留(如同我们在 read 和 write 中见到的), 而一个负值被看作一个错误并且被用来在用户空间设置 errno.[21][21]

"exchange"和"shift"操作对于 scull 没有特别的用处. 我们实现"exchange"来显示驱动如何结合独立的操作到单个的原子的操作, 并且"shift"来连接"tell"和"query". 有时需要象这样的原子的测试-和-设置操作, 特别地, 当应用程序需要设置和释放锁.

命令的明确的序号没有特别的含义. 它只用来区分命令. 实际上, 你甚至可使用相同的序号给一个读命令和一个写命令, 因为实际的 ioctl 号在"方向"位是不同的, 但是你没有理由这样做. 我们选择在任何地方不使用命令的序号除了声明中, 因此我们不分配一个返回值给它. 这就是为什么明确的号出现在之前给定的定义中. 这个例子展示了一个使用命令号的方法, 但是你有自由不这样做.

除了少数几个预定义的命令(马上就讨论), ioctl 的 cmd 参数的值当前不被内核使用, 并且在将来也很不可能. 因此, 你可以, 如果你觉得懒, 避免前面展示的复杂的声明并明确声明一组调整数字. 另一方面, 如果你做了, 你不会从使用这些位段中获益, 并且你会遇到困难如果你曾提交你的代码来包含在主线内核中. 头文件<linux/kd.h> 是这个老式方法的例子, 使用 16-位的调整值来定义 ioctl 命令. 那个源代码依靠调整数因为使用那个时候遵循的惯例, 不是由于懒惰. 现在改变它可能导致无理由的不兼容.

---

[21] [21] 实际上, 所有的当前使用的 libc 实现(包括 uClibc) 仅将 -4095 到 -1 的值当作错误码. 不幸的是, 能够返回大的负数而不是小的, 没有多大用处.

6.1.2. 返回值

ioctl 的实现常常是一个 switch 语句, 基于命令号. 但是当命令号没有匹配一个有效的操作时缺省的选择应当是什么? 这个问题是有争议的. 几个内核函数返回 -ENIVAL("Invalid argument"), 它有意义是因为命令参数确实不是一个有效的. POSIX 标准, 但是, 说如果一个不合适的 ioctl 命令被发出, 那么 -ENOTTY 应当被返回. 这个错误码被 C 库解释为"设备的不适当的 ioctl", 这常常正是程序员需要听到的. 然而, 它仍然是相当普遍的来返回 -EINVAL, 对于响应一个无效的 ioctl 命令.

6.1.3. 预定义的命令

尽管 ioctl 系统调用最常用来作用于设备, 内核能识别几个命令. 注意这些命令, 当用到你的设备时, 在你自己的文件操作被调用之前被解码. 因此, 如果你选择相同的号给一个你的 ioctl 命令, 你不会看到任何的给那个命令的请求, 并且应用程序获得某些不期望的东西, 因为在 ioctl 号之间的冲突.

预定义命令分为 3 类:

- 可对任何文件发出的(常规, 设备, FIFO, 或者 socket) 的那些.
- 只对常规文件发出的那些.
- 对文件系统类型特殊的那些.

最后一类的命令由宿主文件系统的实现来执行(这是 chattr 命令如何工作的). 设备驱动编写者只对第一类命令感兴趣, 它们的魔数是 "T". 查看其他类的工作留给读者作为练习; ext2_ioctl 是最有趣的函数(并且比预期的要容易理解), 因为它实现 append-only 标志和 immutable 标志.

下列 ioctl 命令是预定义给任何文件, 包括设备特殊的文件:

FIOCLEX

> 设置 close-on-exec 标志(File IOctl Close on EXec). 设置这个标志使文件描述符被关闭, 当调用进程执行一个新程序时.

FIONCLEX

> 清除 close-no-exec 标志(File IOctl Not CLose on EXec). 这个命令恢复普通文件行为, 复原上面 FIOCLEX 所做的. FIOASYNC 为这个文件设置或者复位异步通知(如同在本章中"异步通知"一节中讨论的). 注意直到 Linux 2.2.4 版本的内核不正确地使用这个命令来修改 O_SYNC 标志. 因为两个动作都可通过 fcntl 来完成, 没有人真正使用 FIOASYNC 命令, 它在这里出现只是为了完整性.

FIOQSIZE

> 这个命令返回一个文件或者目录的大小; 当用作一个设备文件, 但是, 它返回一个 ENOTTY 错误.

FIONBIO

"File IOctl Non-Blocking I/O"(在"阻塞和非阻塞操作"一节中描述). 这个调用修改在 filp->f_flags 中的 O_NONBLOCK 标志. 给这个系统调用的第 3 个参数用作指示是否这个标志被置位或者清除. (我们将在本章看到这个标志的角色). 注意常用的改变这个标志的方法是使用 fcntl 系统调用, 使用 F_SETFL 命令.

列表中的最后一项介绍了一个新的系统调用, fcntl, 它看来象 ioctl. 事实上, fcntl 调用非常类似 ioctl, 它也是获得一个命令参数和一个额外的(可选地)参数. 它保持和 ioctl 独立主要是因为历史原因: 当 Unix 开发者面对控制 I/O 操作的问题时, 他们决定文件和设备是不同的. 那时, 有 ioctl 实现的唯一设备是 ttys, 它解释了为什么 -ENOTTY 是标准的对不正确 ioctl 命令的回答. 事情已经改变, 但是 fcntl 保留为一个独立的系统调用.

6.1.4. 使用 ioctl 参数

在看 scull 驱动的 ioctl 代码之前, 我们需要涉及的另一点是如何使用这个额外的参数. 如果它是一个整数, 就容易: 它可以直接使用. 如果它是一个指针, 但是, 必须小心些.

当用一个指针引用用户空间, 我们必须确保用户地址是有效的. 试图存取一个没验证过的用户提供的指针可能导致不正确的行为, 一个内核 oops, 系统崩溃, 或者安全问题. 它是驱动的责任来对每个它使用的用户空间地址进行正确的检查, 并且返回一个错误如果它是无效的.

在第 3 章, 我们看了 copy_from_user 和 copy_to_user 函数, 它们可用来安全地移动数据到和从用户空间. 这些函数也可用在 ioctl 方法中, 但是 ioctl 调用常常包含小数据项, 可通过其他方法更有效地操作. 开始, 地址校验(不传送数据)由函数 access_ok 实现, 它定义在 <asm/uaccess.h>:

```
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应当是 VERIFY_READ 或者 VERIFY_WRITE, 依据这个要进行的动作是否是读用户空间内存区或者写它. addr 参数持有一个用户空间地址, size 是一个字节量. 例如, 如果 ioctl 需要从用户空间读一个整数, size 是 sizeof(int). 如果你需要读和写给定地址, 使用 VERIFY_WRITE, 因为它是 VERIRY_READ 的超集.

不象大部分的内核函数, access_ok 返回一个布尔值: 1 是成功(存取没问题)和 0 是失败(存取有问题). 如果它返回假, 驱动应当返回 -EFAULT 给调用者.

关于 access_ok 有多个有趣的东西要注意. 首先, 它不做校验内存存取的完整工作; 它只检查看这个内存引用是在这个进程有合理权限的内存范围中. 特别地, access_ok 确保这个地址不指向内核空间内存. 第 2, 大部分驱动代码不需要真正调用 access_ok. 后面描述的内存存取函数为你负责这个. 但是, 我们来演示它的使用, 以便你可见到它如何完成.

scull 源码利用了 ioclt 号中的位段来检查参数, 在 switch 之前:

```
int err = 0, tmp;
```

```
int retval = 0;
/*
 * extract the type and number bitfields, and don't decode
 * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC)
        return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR)
        return -ENOTTY;

/*
 * the direction is a bitmask, and VERIFY_WRITE catches R/W
 * transfers. `Type' is user-oriented, while
 * access_ok is kernel-oriented, so the concept of "read" and
 * "write" is reversed
 */
if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
if (err)
        return -EFAULT;
```

在调用 access_ok 之后，驱动可安全地进行真正的传输. 加上 copy_from_user 和 copy_to_user_ 函数，程序员可利用一组为被最多使用的数据大小(1, 2, 4, 和 8 字节)而优化过的函数. 这些函数在下面列表中描述，它们定义在 <asm/uaccess.h>:

put_user(datum, ptr)
__put_user(datum, ptr)

> 这些宏定义写 datum 到用户空间; 它们相对快，并且应当被调用来代替 copy_to_user 无论何时要传送单个值时. 这些宏已被编写来允许传递任何类型的指针到 put_user，只要它是一个用户空间地址. 传送的数据大小依赖 prt 参数的类型，并且在编译时使用 sizeof 和 typeof 等编译器内建宏确定. 结果是，如果 prt 是一个 char 指针，传送一个字节，以及对于 2, 4, 和 可能的 8 字节.
>
> put_user 检查来确保这个进程能够写入给定的内存地址. 它在成功时返回 0，并且在错误时返回 -EFAULT. __put_user 进行更少的检查(它不调用 access_ok)，但是仍然能够失败如果被指向的内存对用户是不可写的. 因此，__put_user 应当只用在内存区已经用 access_ok 检查过的时候.
>
> 作为一个通用的规则，当你实现一个 read 方法时，调用 __put_user 来节省几个周期，或者当你拷贝几个项时，因此，在第一次数据传送之前调用 access_ok 一次，如同上面 ioctl 所示.

get_user(local, ptr)
__get_user(local, ptr)

这些宏定义用来从用户空间接收单个数据. 它们象 put_user 和 __put_user, 但是在相反方向传递数据. 获取的值存储于本地变量 local; 返回值指出这个操作是否成功. 再次, __get_user 应当只用在已经使用 access_ok 校验过的地址.

如果做一个尝试来使用一个列出的函数来传送一个不适合特定大小的值, 结果常常是一个来自编译器的奇怪消息, 例如"coversion to non-scalar type requested". 在这些情况中, 必须使用 copy_to_user 或者 copy_from_user.

6.1.5. 兼容性和受限操作

存取一个设备由设备文件上的许可权控制, 并且驱动正常地不涉及到许可权的检查. 但是, 有些情形, 在保证给任何用户对设备的读写许可的地方, 一些控制操作仍然应当被拒绝. 例如, 不是所有的磁带驱动器的用户都应当能够设置它的缺省块大小, 并且一个已经被给予对一个磁盘设备读写权限的用户应当仍然可能被拒绝来格式化它. 在这样的情况下, 驱动必须进行额外的检查来确保用户能够进行被请求的操作.

传统上 unix 系统对超级用户帐户限制了特权操作. 这意味着特权是一个全有-或-全无的东西 -- 超级用户可能任意做任何事情, 但是所有其他的用户被高度限制了. Linux 内核提供了一个更加灵活的系统, 称为能力. 一个基于能力的系统丢弃了全有-或全无模式, 并且打破特权操作为独立的子类. 这种方式, 一个特殊的用户(或者是程序)可被授权来进行一个特定的特权操作而不必泄漏进行其他的, 无关的操作的能力. 内核在许可权管理上排他地使用能力, 并且输出 2 个系统调用 capget 和 capset, 来允许它们被从用户空间管理.

全部能力可在 <linux/capability.h> 中找到. 这些是对系统唯一可用的能力; 对于驱动作者或者系统管理员, 不可能不修改内核源码而来定义新的. 设备驱动编写者可能感兴趣的这些能力的一个子集, 包括下面:

CAP_DAC_OVERRIDE

　　这个能力来推翻在文件和目录上的存取的限制(数据存取控制, 或者 DAC).

CAP_NET_ADMIN

　　进行网络管理任务的能力, 包括那些能够影响网络接口的.

CAP_SYS_MODULE

　　加载或去除内核模块的能力.

CAP_SYS_RAWIO

　　进行 "raw" I/O 操作的能力. 例子包括存取设备端口或者直接和 USB 设备通讯.

CAP_SYS_ADMIN

　　一个捕获-全部的能力, 提供对许多系统管理操作的存取.

CAP_SYS_TTY_CONFIG

进行 tty 配置任务的能力.

在进行一个特权操作之前，一个设备驱动应当检查调用进程有合适的能力；不这样做可能导致用户进程进行非法的操作，对系统的稳定和安全有坏的后果. 能力检查是通过 capable 函数来进行的(定义在 <linux/sched.h>):

```
int capable(int capability);
```

在 scull 例子驱动中，任何用户被许可来查询 quantum 和 quantum 集的大小. 只有特权用户，但是，可改变这些值，因为不适当的值可能很坏地影响系统性能. 当需要时，ioctl 的 scull 实现检查用户的特权级别，如下:

```
if (! capable (CAP_SYS_ADMIN))
return -EPERM;
```

在这个任务缺乏一个更加特定的能力时，CAP_SYS_ADMIN 被选择来做这个测试.

6.1.6. ioctl 命令的实现

ioctl 的 scull 实现只传递设备的配置参数，并且象下面这样容易:

```
switch(cmd)
{
case SCULL_IOCRESET:
        scull_quantum = SCULL_QUANTUM;
        scull_qset = SCULL_QSET;
        break;

case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
        if (! capable (CAP_SYS_ADMIN))

                return -EPERM;
        retval = __get_user(scull_quantum, (int __user *)arg);
        break;

case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
        if (! capable (CAP_SYS_ADMIN))

                return -EPERM;
        scull_quantum = arg;
        break;

case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
        retval = __put_user(scull_quantum, (int __user *)arg);
        break;
```

```
case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
        return scull_quantum;

case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
        if (! capable (CAP_SYS_ADMIN))

                return -EPERM;
        tmp = scull_quantum;
        retval = __get_user(scull_quantum, (int __user *)arg);
        if (retval == 0)

                retval = __put_user(tmp, (int __user *)arg);
        break;

case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
        if (! capable (CAP_SYS_ADMIN))
                return -EPERM;
        tmp = scull_quantum;
        scull_quantum = arg;
        return tmp;

default: /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
}
return retval;
```

scull 还包含 6 个入口项作用于 scull_qset. 这些入口项和给 scull_quantum 的是一致的, 并且不值得展示出来.

从调用者的观点看(即从用户空间), 这 6 种传递和接收参数的方法看来如下:

```
int quantum;
ioctl(fd,SCULL_IOCSQUANTUM, &quantum);   /* Set by pointer */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);   /* Set by value */
ioctl(fd,SCULL_IOCGQUANTUM, &quantum);   /* Get by pointer */
quantum = ioctl(fd,SCULL_IOCQQUANTUM);   /* Get by return value */
ioctl(fd,SCULL_IOCXQUANTUM, &quantum);   /* Exchange by pointer */

quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* Exchange by value */
```

当然, 一个正常的驱动不可能实现这样一个调用模式的混合体. 我们这里这样做只是为了演示做事情的不同方式. 但是, 正常地, 数据交换将一致地进行, 通过指针或者通过值, 并且要避免混合这 2 种技术.

6.1.7. 不用 ioctl 的设备控制

有时控制设备最好是通过写控制序列到设备自身来实现. 例如, 这个技术用在控制台驱动中, 这里所谓的 escape 序列被用来移动光标, 改变缺省的颜色, 或者进行其他的配置任务. 这样实现设备控制的好处是用户可仅仅通过写数据控制设备, 不必使用(或者有时候写)只为配置设备而建立的程序. 当设备可这样来控制, 发出命令的程序甚至常常不需要运行在和它要控制的设备所在的同一个系统上.

例如, setterm 程序作用于控制台(或者其他终端)配置, 通过打印 escape 序列. 控制程序可位于和被控制的设备不同的一台计算机上, 因为一个简单的数据流重定向可完成这个配置工作. 这是每次你运行一个远程 tty 会话时所发生的事情: escape 序列在远端被打印但是影响到本地的 tty; 然而, 这个技术不局限于 ttys.

通过打印来控制的缺点是它给设备增加了策略限制; 例如, 它仅仅当你确信在正常操作时控制序列不会出现在正被写入设备的数据中. 这对于 ttys 只是部分正确的. 尽管一个文本显示意味着只显示 ASCII 字符, 有时控制字符可潜入正被写入的数据中, 并且可能, 因此, 影响控制台的配置. 例如, 这可能发生在你显示一个二进制文件到屏幕时; 产生的乱码可能包含任何东西, 并且最后你常常在你的控制台上出现错误的字体.

通过写来控制是当然的使用方法了, 对于不用传送数据而只是响应命令的设备, 例如遥控设备.

例如, 被你们作者当中的一个编写来好玩的驱动, 移动一个 2 轴上的摄像机. 在这个驱动里, 这个"设备"是一对老式步进电机, 它们不能真正读或写. 给一个步进电机"发送数据流"的概念没有任何意义. 在这个情况下, 驱动解释正被写入的数据作为 ASCII 命令并且转换这个请求为脉冲序列, 来操纵步进电机. 这个概念类似于, 有些, 你发给猫的 AT 命令来建立通讯, 主要的不同是和猫通讯的串口必须也传送真正的数据. 直接设备控制的好处是你可以使用 cat 来移动摄像机, 而不必写和编译特殊的代码来发出 ioctl 调用.

当编写面向命令的驱动, 没有理由实现 ioctl 命令. 一个解释器中的额外命令更容易实现并使用.

有时, 然而, 你可能选择使用其他的方法:不必转变 write 方法为一个解释器和避免 ioctl, 你可能选择完全避免写并且专门使用 ioctl 命令, 而实现驱动为使用一个特殊的命令行工具来发送这些命令到驱动. 这个方法转移复杂性从内核空间到用户空间, 这里可能更易处理, 并且帮助保持驱动小, 而拒绝使用简单的 cat 或者 echo 命令.

6.2. 阻塞 I/O

回顾第 3 章, 我们看到如何实现 read 和 write 方法. 在此, 但是, 我们跳过了一个重要的问题:一个驱动当它无法立刻满足请求应当如何响应? 一个对 read 的调用可能当没有数据时到来, 而以后会期待更多的数据. 或者一个进程可能试图写, 但是你的设备没有准备好接受数据, 因为你的输出缓冲满了. 调用进程往往不关心这种问题; 程序员只希望调用 read 或 write 并且使调用返回, 在必要的工作已完成后. 这样, 在这样的情形中, 你的驱动应当(缺省地)阻塞进程, 使它进入睡眠直到请求可继续.

本节展示如何使一个进程睡眠并且之后再次唤醒它. 如常, 但是, 我们必须首先解释几个概念.

6.2.1. 睡眠的介绍

对于一个进程"睡眠"意味着什么? 当一个进程被置为睡眠, 它被标识为处于一个特殊的状态并且从调度器的运行队列中去除. 直到发生某些事情改变了那个状态, 这个进程将不被在任何 CPU 上调度, 并且, 因此, 将不会运行. 一个睡着的进程已被搁置到系统的一边, 等待以后发生事件.

对于一个 Linux 驱动使一个进程睡眠是一个容易做的事情. 但是, 有几个规则必须记住以安全的方式编码睡眠.

这些规则的第一个是: 当你运行在原子上下文时不能睡眠. 我们在第 5 章介绍过原子操作; 一个原子上下文只是一个状态, 这里多个步骤必须在没有任何类型的并发存取的情况下进行. 这意味着, 对于睡眠, 是你的驱动在持有一个自旋锁, seqlock, 或者 RCU 锁时不能睡眠. 如果你已关闭中断你也不能睡眠. 在持有一个旗标时睡眠是合法的, 但是你应当仔细查看这样做的任何代码. 如果代码在持有一个旗标时睡眠, 任何其他的等待这个旗标的线程也睡眠. 因此发生在持有旗标时的任何睡眠应当短暂, 并且你应当说服自己, 由于持有这个旗标, 你不能阻塞这个将最终唤醒你的进程.

另一件要记住的事情是, 当你醒来, 你从不知道你的进程离开 CPU 多长时间或者同时已经发生了什么改变. 你也常常不知道是否另一个进程已经睡眠等待同一个事件; 那个进程可能在你之前醒来并且获取了你在等待的资源. 结果是你不能关于你醒后的系统状态做任何的假设, 并且你必须检查来确保你在等待的条件是, 确实, 真的.

一个另外的相关的点, 当然, 是你的进程不能睡眠除非确信其他人, 在某处的, 将唤醒它. 做唤醒工作的代码必须也能够找到你的进程来做它的工作. 确保一个唤醒发生, 是深入考虑你的代码和对于每次睡眠, 确切知道什么系列的事件将结束那次睡眠. 使你的进程可能被找到, 真正地, 通过一个称为等待队列的数据结构实现的. 一个等待队列就是它听起来的样子:一个进程列表, 都等待一个特定的事件.

在 Linux 中, 一个等待队列由一个"等待队列头"来管理, 一个 wait_queue_head_t 类型的结构, 定义在<linux/wait.h>中. 一个等待队列头可被定义和初始化, 使用:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地, 如下:

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

我们将很快返回到等待队列结构, 但是我们知道了足够多的来首先看看睡眠和唤醒.

6.2.2. 简单睡眠

当一个进程睡眠, 它这样做以期望某些条件在以后会成真. 如我们之前注意到的, 任何睡眠的进程必须在它再次醒来时检查来确保它在等待的条件真正为真. Linux 内核中睡眠的最简单方式是一个宏定义, 称为 wait_event(有几个变体); 它结合了处理睡眠的细节和进程在等待的条件的检查. wait_event 的形式是:

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

在所有上面的形式中, queue 是要用的等待队列头. 注意它是"通过值"传递的. 条件是一个被这个宏在睡眠前后所求值的任意的布尔表达式; 直到条件求值为真值, 进程继续睡眠. 注意条件可能被任意次地求值, 因此它不应当有任何边界效应.

如果你使用 wait_event, 你的进程被置为不可中断地睡眠, 如同我们之前已经提到的, 它常常不是你所要的. 首选的选择是 wait_event_interruptible, 它可能被信号中断. 这个版本返回一个你应当检查的整数值; 一个非零值意味着你的睡眠被某些信号打断, 并且你的驱动可能应当返回 -ERESTARTSYS. 最后的版本(wait_event_timeout 和 wait_event_interruptible_timeout)等待一段有限的时间; 在这个时间期间(以嘀哒数表达的, 我们将在第 7 章讨论)超时后, 这个宏返回一个 0 值而不管条件是如何求值的.

图片的另一半, 当然, 是唤醒. 一些其他的执行线程(一个不同的进程, 或者一个中断处理, 也许)必须为你进行唤醒, 因为你的进程, 当然, 是在睡眠. 基本的唤醒睡眠进程的函数称为 wake_up. 它有几个形式(但是我们现在只看其中 2 个):

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 唤醒所有的在给定队列上等待的进程(尽管这个情形比那个要复杂一些, 如同我们之后将见到的). 其他的形式(wake_up_interruptible)限制它自己到处理一个可中断的睡眠. 通常, 这 2 个是不用区分的(如果你使用可中断的睡眠); 实际上, 惯例是使用 wake_up 如果你在使用 wait_event , wake_up_interruptible 如果你在使用 wait_event_interruptible.

我们现在知道足够多来看一个简单的睡眠和唤醒的例子. 在这个例子代码中, 你可找到一个称为 sleepy 的模块. 它实现一个有简单行为的设备:任何试图从这个设备读取的进程都被置为睡眠. 无论何时一个进程写这个设备, 所有的睡眠进程被唤醒. 这个行为由下面的 read 和 write 方法实现:

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read (struct file *filp, char __user *buf, size_t count, loff_t
*pos)
{
        printk(KERN_DEBUG "process %i (%s) going to sleep\n",
               current->pid, current->comm);
        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
        return 0; /* EOF */
}
```

```
ssize_t sleepy_write (struct file *filp, const char __user *buf, size_t count,
loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
               current->pid, current->comm);
        flag = 1;
        wake_up_interruptible(&wq);
        return count; /* succeed, to avoid retrial */

}
```

注意这个例子里 flag 变量的使用. 因为 wait_event_interruptible 检查一个必须变为真的条件, 我们使用 flag 来创建那个条件.

有趣的是考虑当 sleepy_write 被调用时如果有 2 个进程在等待会发生什么. 因为 sleepy_read 重置 flag 为 0 一旦它醒来, 你可能认为醒来的第 2 个进程会立刻回到睡眠. 在一个单处理器系统, 这几乎一直是发生的事情. 但是重要的是要理解为什么你不能依赖这个行为. wake_up_interruptible 调用将使 2 个睡眠进程醒来. 完全可能它们都注意到 flag 是非零, 在另一个有机会重置它之前. 对于这个小模块, 这个竞争条件是不重要的. 在一个真实的驱动中, 这种竞争可能导致少见的难于查找的崩溃. 如果正确的操作要求只能有一个进程看到这个非零值, 它将必须以原子的方式被测试. 我们将见到一个真正的驱动如何处理这样的情况. 但首先我们必须开始另一个主题.

6.2.3. 阻塞和非阻塞操作

在我们看全功能的 read 和 write 方法的实现之前, 我们触及的最后一点是决定何时使进程睡眠. 有时实现正确的 unix 语义要求一个操作不阻塞, 即便它不能完全地进行下去.

有时还有调用进程通知你他不想阻塞, 不管它的 I/O 是否继续. 明确的非阻塞 I/O 由 filp->f_flags 中的 O_NONBLOCK 标志来指示. 这个标志定义于 <linux/fcntl.h>, 被 <linux/fs.h>自动包含. 这个标志得名自"打开-非阻塞", 因为它可在打开时指定(并且起初只能在那里指定). 如果你浏览源码, 你会发现一些对一个 O_NDELAY 标志的引用; 这是一个替代 O_NONBLOCK 的名子, 为兼容 System V 代码而被接受的. 这个标志缺省地被清除, 因为一个等待数据的进程的正常行为仅仅是睡眠. 在一个阻塞操作的情况下, 这是缺省地, 下列的行为应当实现来符合标准语法:

- 如果一个进程调用 read 但是没有数据可用(尚未), 这个进程必须阻塞. 这个进程在有数据达到时被立刻唤醒, 并且那个数据被返回给调用者, 即便小于在给方法的 count 参数中请求的数量.
- 如果一个进程调用 write 并且在缓冲中没有空间, 这个进程必须阻塞, 并且它必须在一个与用作 read 的不同的等待队列中. 当一些数据被写入硬件设备, 并且在输出缓冲中的空间变空闲, 这个进程被唤醒并且写调用成功, 尽管数据可能只被部分写入如果在缓冲只没有空间给被请求的 count 字节.

这 2 句都假定有输入和输出缓冲; 实际上, 几乎每个设备驱动都有. 要求有输入缓冲是为了避免丢失到达的数据, 当无人在读时. 相反, 数据在写时不能丢失, 因为如果系统调

用不能接收数据字节，它们保留在用户空间缓冲. 即便如此，输出缓冲几乎一直有用，对于从硬件挤出更多的性能.

在驱动中实现输出缓冲所获得的性能来自减少了上下文切换和用户级/内核级切换的次数. 没有一个输出缓冲(假定一个慢速设备)，每次系统调用接收这样一个或几个字符，并且当一个进程在 write 中睡眠，另一个进程运行(那是一次上下文切换). 当第一个进程被唤醒，它恢复(另一次上下文切换)，写返回(内核/用户转换)，并且这个进程重新发出系统调用来写入更多的数据(用户/内核转换)；这个调用阻塞并且循环继续. 增加一个输出缓冲可允许驱动在每个写调用中接收大的数据块，性能上有相应的提高. 如果这个缓冲足够大，写调用在第一次尝试就成功 -- 被缓冲的数据之后将被推到设备 -- 不必控制需要返回用户空间来第二次或者第三次写调用. 选择一个合适的值给输出缓冲显然是设备特定的.

我们不使用一个输入缓冲在 scull 中，因为数据当发出 read 时已经可用. 类似的，不用输出缓冲，因为数据被简单地拷贝到和设备关联的内存区. 本质上，这个设备是一个缓冲，因此额外缓冲的实现可能是多余的. 我们将在第 10 章见到缓冲的使用.

如果指定 O_NONBLOCK，read 和 write 的行为是不同的. 在这个情况下，这个调用简单地返回 -EAGAIN(("try it agin")如果一个进程当没有数据可用时调用 read ，或者如果当缓冲中没有空间时它调用 write .

如你可能期望的，非阻塞操作立刻返回，允许这个应用程序轮询数据. 应用程序当使用 stdio 函数处理非阻塞文件中，必须小心，因为它们容易搞错一个的非阻塞返回为 EOF. 它们始终必须检查 errno.

自然地，O_NONBLOCK 也在 open 方法中有意义. 这个发生在当这个调用真正阻塞长时间时；例如，当打开(为读存取)一个 没有写者的(尚无)FIFO，或者存取一个磁盘文件使用一个悬挂锁. 常常地，打开一个设备或者成功或者失败，没有必要等待外部的事件. 有时，但是，打开这个设备需要一个长的初始化，并且你可能选择在你的 open 方法中支持 O_NONBLOCK ，通过立刻返回 -EAGAIN,如果这个标志被设置. 在开始这个设备的初始化进程之后. 这个驱动可能还实现一个阻塞 open 来支持存取策略，通过类似于文件锁的方式. 我们将见到这样一个实现在"阻塞 open 作为对 EBUSY 的替代"一节，在本章后面.

一些驱动可能还实现特别的语义给 O_NONBLOCK；例如，一个磁带设备的 open 常常阻塞直到插入一个磁带. 如果这个磁带驱动器使用 O_NONBLOCK 打开，这个 open 立刻成功，不管是否介质在或不在.

只有 read，write，和 open 文件操作受到非阻塞标志影响.

6.2.4. 一个阻塞 I/O 的例子

最后，我们看一个实现了阻塞 I/O 的真实驱动方法的例子. 这个例子来自 scullpipe 驱动；它是 scull 的一个特殊形式，实现了一个象管道的设备.

在驱动中，一个阻塞在读调用上的进程被唤醒，当数据到达时；常常地硬件发出一个中断来指示这样一个事件，并且驱动唤醒等待的进程作为处理这个中断的一部分. scullpipe 驱动不同，以至于它可运行而不需要任何特殊的硬件或者一个中断处理. 我们选择来使用

另一个进程来产生数据并唤醒读进程；类似地，读进程被用来唤醒正在等待缓冲空间可用的写者进程.

这个设备驱动使用一个设备结构，它包含 2 个等待队列和一个缓冲. 缓冲大小是以常用的方法可配置的(在编译时间，加载时间，或者运行时间).

```
struct scull_pipe
{
        wait_queue_head_t inq, outq; /* read and write queues */
        char *buffer, *end; /* begin of buf, end of buf */
        int buffersize; /* used in pointer arithmetic */
        char *rp, *wp; /* where to read, where to write */
        int nreaders, nwriters; /* number of openings for r/w */
        struct fasync_struct *async_queue; /* asynchronous readers */
        struct semaphore sem;  /* mutual exclusion semaphore */
        struct cdev cdev;  /* Char device structure */
};
```

读实现既管理阻塞也管理非阻塞输入，看来如此:

```
static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t count,
loff_t *f_pos)
{
        struct scull_pipe *dev = filp->private_data;
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        while (dev->rp == dev->wp)
        { /* nothing to read */
                up(&dev->sem); /* release the lock */
                if (filp->f_flags & O_NONBLOCK)

                        return -EAGAIN;
                PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
                if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
                        return -ERESTARTSYS; /* signal: tell the fs layer to
handle it */ /* otherwise loop, but first reacquire the lock */
                if (down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
        }
        /* ok, data is there, return something */

        if (dev->wp > dev->rp)
                count = min(count, (size_t)(dev->wp - dev->rp));
        else /* the write pointer has wrapped, return data up to dev->end */
                count = min(count, (size_t)(dev->end - dev->rp));
        if (copy_to_user(buf, dev->rp, count))
```

```
        {
                up (&dev->sem);
                return -EFAULT;
        }
        dev->rp += count;
        if (dev->rp == dev->end)

                dev->rp = dev->buffer; /* wrapped */
        up (&dev->sem);

        /* finally, awake any writers and return */
        wake_up_interruptible(&dev->outq);
        PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
        return count;
}
```

如同你可见的，我们在代码中留有一些 PDEBUG 语句. 当你编译这个驱动，你可使能消息机制来易于跟随不同进程间的交互.

让我们仔细看看 scull_p_read 如何处理对数据的等待. 这个 while 循环在持有设备旗标下测试这个缓冲. 如果有数据在那里，我们知道我们可立刻返回给用户，不必睡眠，因此整个循环被跳过. 相反，如果这个缓冲是空的，我们必须睡眠. 但是在我们可做这个之前，我们必须丢掉设备旗标；如果我们要持有它而睡眠，就不会有写者有机会唤醒我们. 一旦这个确保被丢掉，我们做一个快速检查来看是否用户已请求非阻塞 I/O，并且如果是这样就返回. 否则，是时间调用 wait_event_interruptible.

一旦我们过了这个调用，某些东东已经唤醒了我们，但是我们不知道是什么. 一个可能是进程接收到了一个信号. 包含 wait_event_interruptible 调用的这个 if 语句检查这种情况. 这个语句保证了正确的和被期望的对信号的反应，它可能负责唤醒这个进程(因为我们处于一个可中断的睡眠). 如果一个信号已经到达并且它没有被这个进程阻塞，正确的做法是让内核的上层处理这个事件. 到此，这个驱动返回 -ERESTARTSYS 到调用者；这个值被虚拟文件系统(VFS)在内部使用，它或者重启系统调用或者返回 -EINTR 给用户空间. 我们使用相同类型的检查来处理信号，给每个读和写实现.

但是，即便没有一个信号，我们还是不确切知道有数据在那里为获取. 其他人也可能已经在等待数据，并且它们可能赢得竞争并且首先得到数据. 因此我们必须再次获取设备旗标；只有这时我们才可以测试读缓冲(在 while 循环中)并且真正知道我们可以返回缓冲中的数据给用户. 全部这个代码的最终结果是，当我们从 while 循环中退出时，我们知道旗标被获得并且缓冲中有数据我们可以用.

仅仅为了完整，我们要注意，scull_p_read 可以在另一个地方睡眠，在我们获得设备旗标之后：对 copy_to_user 的调用. 如果 scull 当在内核和用户空间之间拷贝数据时睡眠，它在持有设备旗标中睡眠. 在这种情况下持有旗标是合理的因为它不能死锁系统(我们知道内核将进行拷贝到用户空间并且在不加锁进程中的同一个旗标下唤醒我们)，并且因为重要的是设备内存数组在驱动睡眠时不改变.

6.2.5. 高级睡眠

许多驱动能够满足它们的睡眠要求，使用至今我们已涉及到的函数. 但是，有时需要深入理解 Linux 等待队列机制如何工作. 复杂的加锁或者性能需要可强制一个驱动来使用低层函数来影响一个睡眠. 在本节，我们在低层看而理解在一个进程睡眠时发生了什么.

6.2.5.1. 一个进程如何睡眠

如果我们深入 <linux/wait.h>，你见到在 wait_queue_head_t 类型后面的数据结构是非常简单的；它包含一个自旋锁和一个链表. 这个链表是一个等待队列入口，它被声明做 wait_queue_t. 这个结构包含关于睡眠进程的信息和它想怎样被唤醒.

使一个进程睡眠的第一步常常是分配和初始化一个 wait_queue_t 结构，随后将其添加到正确的等待队列. 当所有东西都就位了，负责唤醒工作的人就可以找到正确的进程.

下一步是设置进程的状态来标志它为睡眠. 在 <linux/sched.h> 中定义有几个任务状态. TASK_RUNNING 意思是进程能够运行，尽管不必在任何特定的时刻在处理器上运行. 有 2 个状态指示一个进程是在睡眠：TASK_INTERRUPTIBLE 和 TASK_UNTINTERRUPTIBLE；当然，它们对应 2 类的睡眠. 其他的状态正常地和驱动编写者无关.

在 2.6 内核，对于驱动代码通常不需要直接操作进程状态. 但是，如果你需要这样做，使用的代码是：

```
void set_current_state(int new_state);
```

在老的代码中，你常常见到如此的东西：

```
current->state = TASK_INTERRUPTIBLE;
```

但是象这样直接改变 current 是不鼓励的；当数据结构改变时这样的代码会轻易地失效. 但是，上面的代码确实展示了自己改变一个进程的当前状态不能使其睡眠. 通过改变 current 状态，你已改变了调度器对待进程的方式，但是你还未让出处理器.

放弃处理器是最后一步，但是要首先做一件事：你必须先检查你在睡眠的条件. 做这个检查失败会引入一个竞争条件；如果在你忙于上面的这个过程并且有其他的线程刚刚试图唤醒你，如果这个条件变为真会发生什么？你可能错过唤醒并且睡眠超过你预想的时间. 因此，在睡眠的代码下面，典型地你会见到下面的代码：

```
if (!condition)
    schedule();
```

通过在设置了进程状态后检查我们的条件，我们涵盖了所有的可能的事件进展. 如果我们在等待的条件已经在设置进程状态之前到来，我们在这个检查中注意到并且不真正地睡眠. 如果之后发生了唤醒，进程被置为可运行的不管是否我们已真正进入睡眠.

调用 schedule ，当然，是引用调度器和让出 CPU 的方式. 无论何时你调用这个函数，你是在告诉内核来考虑应当运行哪个进程并且转换控制到那个进程，如果必要. 因此你从不知道在 schedule 返回到你的代码会是多长时间.

在 if 测试和可能的调用 schedule (并从其返回)之后, 有些清理工作要做. 因为这个代码不再想睡眠, 它必须保证任务状态被重置为 TASK_RUNNING. 如果代码只是从 schedule 返回, 这一步是不必要的; 那个函数不会返回直到进程处于可运行态. 如果由于不再需要睡眠而对 schedule 的调用被跳过, 进程状态将不正确. 还有必要从等待队列中去除这个进程, 否则它可能被多次唤醒.

6.2.5.2. 手动睡眠

在 Linux 内核的之前的版本, 正式的睡眠要求程序员手动处理所有上面的步骤. 它是一个繁琐的过程, 包含相当多的易出错的样板式的代码. 程序员如果愿意还是可能用那种方式手动睡眠; <linux/sched.h> 包含了所有需要的定义, 以及围绕例子的内核源码. 但是, 有一个更容易的方式.

第一步是创建和初始化一个等待队列. 这常常由这个宏定义完成:

```
DEFINE_WAIT(my_wait);
```

其中, name 是等待队列入口项的名子. 你可用 2 步来做:

```
wait_queue_t my_wait;
init_wait(&my_wait);
```

但是常常更容易的做法是放一个 DEFINE_WAIT 行在循环的顶部, 来实现你的睡眠.

下一步是添加你的等待队列入口到队列, 并且设置进程状态. 2 个任务都由这个函数处理:

```
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
```

这里, queue 和 wait 分别地是等待队列头和进程入口. state 是进程的新状态; 它应当或者是 TASK_INTERRUPTIBLE(给可中断的睡眠, 这常常是你所要的)或者 TASK_UNINTERRUPTIBLE(给不可中断睡眠).

在调用 prepare_to_wait 之后, 进程可调用 schedule -- 在它已检查确认它仍然需要等待之后. 一旦 schedule 返回, 就到了清理时间. 这个任务, 也, 被一个特殊的函数处理:

```
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

之后, 你的代码可测试它的状态并且看是否它需要再次等待.

我们早该需要一个例子了. 之前我们看了 给 scullpipe 的 read 方法, 它使用 wait_event. 同一个驱动中的 write 方法使用 prepare_to_wait 和 finish_wait 来实现它的等待. 正常地, 你不会在一个驱动中象这样混用各种方法, 但是我们这样作是为了能够展示 2 种处理睡眠的方式.

为完整起见, 首先, 我们看 write 方法本身:

```
/* How much space is free? */
```

```
static int spacefree(struct scull_pipe *dev)
{

        if (dev->rp == dev->wp)
                return dev->buffersize - 1;
        return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}

static ssize_t scull_p_write(struct file *filp, const char __user *buf, size_t
count,
                             loff_t *f_pos)
{

        struct scull_pipe *dev = filp->private_data;
        int result;
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        /* Make sure there's space to write */
        result = scull_getwritespace(dev, filp);
        if (result)
                return result; /* scull_getwritespace called up(&dev->sem) */
        /* ok, space is there, accept something */
        count = min(count, (size_t)spacefree(dev));
        if (dev->wp >= dev->rp)
                count = min(count, (size_t)(dev->end - dev->wp)); /* to end-
of-buf */
        else /* the write pointer has wrapped, fill up to rp-1 */
                count = min(count, (size_t)(dev->rp - dev->wp - 1));
        PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev-
>wp, buf);
        if (copy_from_user(dev->wp, buf, count))
        {
                up (&dev->sem);
                return -EFAULT;
        }
        dev->wp += count;
        if (dev->wp == dev->end)
                dev->wp = dev->buffer; /* wrapped */
        up(&dev->sem);

        /* finally, awake any reader */
        wake_up_interruptible(&dev->inq); /* blocked in read() and select() */

        /* and signal asynchronous readers, explained late in chapter 5 */
        if (dev->async_queue)
```

```
                kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
        PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
        return count;
}
```

这个代码看来和 read 方法类似, 除了我们已经将睡眠代码放到了一个单独的函数, 称为 scull_getwritespace. 它的工作是确保在缓冲中有空间给新的数据, 睡眠直到有空间可用. 一旦空间在, scull_p_write 可简单地拷贝用户的数据到那里, 调整指针, 并且唤醒可能已经在等待读取数据的进程.

处理实际的睡眠的代码是:

```
/* Wait for space for writing; caller must hold device semaphore. On
 * error the semaphore will be released before returning. */
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{

        while (spacefree(dev) == 0)
        { /* full */
                DEFINE_WAIT(wait);

                up(&dev->sem);
                if (filp->f_flags & O_NONBLOCK)
                        return -EAGAIN;

                PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
                prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
                if (spacefree(dev) == 0)
                        schedule();
                finish_wait(&dev->outq, &wait);
                if (signal_pending(current))

                        return -ERESTARTSYS; /* signal: tell the fs layer to
handle it */
                if (down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
        }
        return 0;

}
```

再次注意 while 循环. 如果有空间可用而不必睡眠, 这个函数简单地返回. 否则, 它必须丢掉设备旗标并且等待. 这个代码使用 DEFINE_WAIT 来设置一个等待队列入口并且 prepare_to_wait 来准备好实际的睡眠. 接着是对缓冲的必要的检查; 我们必须处理的情况是在我们已经进入 while 循环后以及在我们将自己放入等待队列之前 (并且丢弃了旗标), 缓冲中有空间可用了. 没有这个检查, 如果读进程能够在那时完全清空缓冲, 我们

可能错过我们能得到的唯一的唤醒并且永远睡眠. 在说服我们自己必须睡眠之后, 我们调用 schedule.

值得再看看这个情况: 当睡眠发生在 if 语句测试和调用 schedule 之间, 会发生什么? 在这个情况里, 都好. 这个唤醒重置了进程状态为 TASK_RUNNING 并且 schedule 返回 -- 尽管不必马上. 只要这个测试发生在进程放置自己到等待队列和改变它的状态之后, 事情都会顺利.

为了结束, 我们调用 finish_wait. 对 signal_pending 的调用告诉我们是否我们被一个信号唤醒; 如果是, 我们需要返回到用户并且使它们稍后再试. 否则, 我们请求旗标, 并且再次照常测试空闲空间.

6.2.5.3. 互斥等待

我们已经见到当一个进程调用 wake_up 在等待队列上, 所有的在这个队列上等待的进程被置为可运行的. 在许多情况下, 这是正确的做法. 但是, 在别的情况下, 可能提前知道只有一个被唤醒的进程将成功获得需要的资源, 并且其余的将简单地再次睡眠. 每个这样的进程, 但是, 必须获得处理器, 竞争资源(和任何的管理用的锁), 并且明确地回到睡眠. 如果在等待队列中的进程数目大, 这个"惊群"行为可能严重降低系统的性能.

为应对实际世界中的惊群问题, 内核开发者增加了一个"互斥等待"选项到内核中. 一个互斥等待的行为非常象一个正常的睡眠, 有 2 个重要的不同:

- 当一个等待队列入口有 WQ_FLAG_EXCLUSEVE 标志置位, 它被添加到等待队列的尾部. 没有这个标志的入口项, 相反, 添加到开始.
- 当 wake_up 被在一个等待队列上调用, 它在唤醒第一个有 WQ_FLAG_EXCLUSIVE 标志的进程后停止.

最后的结果是进行互斥等待的进程被一次唤醒一个, 以顺序的方式, 并且没有引起惊群问题. 但内核仍然每次唤醒所有的非互斥等待者.

在驱动中采用互斥等待是要考虑的, 如果满足 2 个条件: 你希望对资源的有效竞争, 并且唤醒一个进程就足够来完全消耗资源当资源可用时. 互斥等待对 Apacheweb 服务器工作地很好, 例如; 当一个新连接进入, 确实地系统中的一个 Apache 进程应当被唤醒来处理它. 我们在 scullpipe 驱动中不使用互斥等待, 但是; 很少见到竞争数据的读者(或者竞争缓冲空间的写者), 并且我们无法知道一个读者, 一旦被唤醒, 将消耗所有的可用数据.

使一个进程进入可中断的等待, 是调用 prepare_to_wait_exclusive 的简单事情:

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait,
int state);
```

这个调用, 当用来代替 prepare_to_wait, 设置"互斥"标志在等待队列入口项并且添加这个进程到等待队列的尾部. 注意没有办法使用 wait_event 和它的变体来进行互斥等待.

6.2.5.4. 唤醒的细节

我们已展现的唤醒进程的样子比内核中真正发生的要简单. 当进程被唤醒时产生的真正动作是被位于等待队列入口项的一个函数控制的. 缺省的唤醒函数[22][22]设置进程为可运行的状态, 并且可能地进行一个上下文切换到有更高优先级进程. 设备驱动应当从不需要提供一个不同的唤醒函数; 如果你例外, 关于如何做的信息见 <linux/wait.h>

我们尚未看到所有的 wake_up 变体. 大部分驱动编写者从不需要其他的, 但是, 为完整起见, 这里是整个集合:

```
wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 唤醒队列中的每个不是在互斥等待中的进程, 并且就只一个互斥等待者, 如果它存在. wake_up_interruptible 同样, 除了它跳过处于不可中断睡眠的进程. 这些函数, 在返回之前, 使一个或多个进程被唤醒来被调度(尽管如果它们被从一个原子上下文调用, 这就不会发生).

```
wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);
```

这些函数类似 wake_up, 除了它们能够唤醒多达 nr 个互斥等待者, 而不只是一个. 注意传递 0 被解释为请求所有的互斥等待者都被唤醒, 而不是一个没有.

```
wake_up_all(wait_queue_head_t *queue);
wake_up_interruptible_all(wait_queue_head_t *queue);
```

这种 wake_up 唤醒所有的进程, 不管它们是否进行互斥等待(尽管可中断的类型仍然跳过在做不可中断等待的进程)

```
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

正常地, 一个被唤醒的进程可能抢占当前进程, 并且在 wake_up 返回之前被调度到处理器. 换句话说, 调用 wake_up 可能不是原子的. 如果调用 wake_up 的进程运行在原子上下文(它可能持有一个自旋锁, 例如, 或者是一个中断处理), 这个重调度不会发生. 正常地, 那个保护是足够的. 但是, 如果你需要明确要求不要被调度出处理器在那时, 你可以使用 wake_up_interruptible 的"同步"变体. 这个函数最常用在当调用者要无论如何重新调度, 并且它会更有效的来首先简单地完成剩下的任何小的工作.

如果上面的全部内容在第一次阅读时没有完全清楚, 不必担心. 很少请求会需要调用 wake_up_interruptible 之外的.

---

[22] [22] 它有一个想象的名子 default_wake_function.

6.2.5.5. 以前的历史: sleep_on

如果你花些时间深入内核源码, 你可能遇到我们到目前忽略讨论的 2 个函数:

```
void sleep_on(wait_queue_head_t *queue);
void interruptible_sleep_on(wait_queue_head_t *queue);
```

如你可能期望的, 这些函数无条件地使当前进程睡眠在给定队列尚. 这些函数强烈不推荐, 但是, 并且你应当从不使用它们. 如果你想想它则问题是明显的: sleep_on 没提供方法来避免竞争条件. 常常有一个窗口在当你的代码决定它必须睡眠时和当 sleep_on 真正影响到睡眠时. 在那个窗口期间到达的唤醒被错过. 因此, 调用 sleep_on 的代码从不是完全安全的.

当前计划对 sleep_on 和 它的变体的调用(有多个我们尚未展示的超时的类型)在不太远的将来从内核中去掉.

6.2.6. 测试 scullpipe 驱动

我们已经见到了 scullpipe 驱动如何实现阻塞 I/O. 如果你想试一试, 这个驱动的源码可在剩下的本书例子中找到. 阻塞 I/O 的动作可通过打开 2 个窗口见到. 第一个可运行一个命令诸如 cat /dev/scullpipe. 如果你接着, 在另一个窗口拷贝文件到 /dev/scullpipe, 你可见到文件的内容出现在第一个窗口.

测试非阻塞的动作是技巧性的, 因为可用于 shell 的传统的程序不做非阻塞操作. misc-progs 源码目录包含下面简单的程序, 称为 nbtest, 来测试非阻塞操作. 所有它做的是拷贝它的输入到它的输出, 使用非阻塞 I/O 和在重试间延时. 延时时间在命令行被传递被缺省是 1 秒.

```
int main(int argc, char **argv)
{

        int delay = 1, n, m = 0;
        if (argc > 1)
                delay=atoi(argv[1]);
        fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
        fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */

        while (1) {
                n = read(0, buffer, 4096);
                if (n >= 0)
                        m = write(1, buffer, n);
                if ((n < 0 || m < 0) && (errno != EAGAIN))
                        break;
                sleep(delay);
        }
        perror(n < 0 ? "stdin" : "stdout");
```

```
        exit(1);
}
```

如果你在一个进程跟踪工具, 如 strace 下运行这个程序, 你可见到每个操作的成功或者失败, 依赖是否当进行操作时有数据可用.

6.3. poll 和 select

使用非阻塞 I/O 的应用程序常常使用 poll, select, 和 epoll 系统调用. poll, select 和 epoll 本质上有相同的功能: 每个允许一个进程来决定它是否可读或者写一个或多个文件而不阻塞. 这些调用也可阻塞进程直到任何一个给定集合的文件描述符可用来读或写. 因此, 它们常常用在必须使用多输入输出流的应用程序, 而不必粘连在它们任何一个上. 相同的功能常常由多个函数提供, 因为 2 个是由不同的团队在几乎相同时间完成的: select 在 BSD Unix 中引入, 而 poll 是 System V 的解决方案. epoll 调用[23][23] 添加在 2.5.45, 作为使查询函数扩展到几千个文件描述符的方法.

支持任何一个这些调用都需要来自设备驱动的支持. 这个支持(对所有 3 个调用)由驱动的 poll 方法调用. 这个方法由下列的原型:

```
unsigned int (*poll) (struct file *filp, poll_table *wait);
```

这个驱动方法被调用, 无论何时用户空间程序进行一个 poll, select, 或者 epoll 系统调用, 涉及一个和驱动相关的文件描述符. 这个设备方法负责这 2 步:

- 1. 在一个或多个可指示查询状态变化的等待队列上调用 poll_wait. 如果没有文件描述符可用作 I/O, 内核使这个进程在等待队列上等待所有的传递给系统调用的文件描述符.
- 2. 返回一个位掩码, 描述可能不必阻塞就立刻进行的操作.

这 2 个操作常常是直接的, 并且趋向与各个驱动看起来类似. 但是, 它们依赖只能由驱动提供的信息, 因此, 必须由每个驱动单独实现.

poll_table 结构, 给 poll 方法的第 2 个参数, 在内核中用来实现 poll, select, 和 epoll 调用; 它在 <linux/poll.h>中声明, 这个文件必须被驱动源码包含. 驱动编写者不必要知道所有它内容并且必须作为一个不透明的对象使用它; 它被传递给驱动方法以便驱动可用每个能唤醒进程的等待队列来加载它, 并且可改变 poll 操作状态. 驱动增加一个等待队列到 poll_table 结构通过调用函数 poll_wait:

```
 void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

poll 方法的第 2 个任务是返回位掩码, 它描述哪个操作可马上被实现; 这也是直接的. 例如, 如果设备有数据可用, 一个读可能不必睡眠而完成; poll 方法应当指示这个时间状态. 几个标志(通过 <linux/poll.h> 定义)用来指示可能的操作:

---

[23] [23] 实际上, epoll 是一组 3 个调用, 都可用来获得查询功能. 但是, 由于我们的目的, 我们可认为它是一个调用.

POLLIN

如果设备可被不阻塞地读，这个位必须设置.

POLLRDNORM

这个位必须设置，如果"正常"数据可用来读. 一个可读的设备返回(POLLIN|POLLRDNORM).

POLLRDBAND

这个位指示带外数据可用来从设备中读取. 当前只用在 Linux 内核的一个地方(DECnet 代码)并且通常对设备驱动不可用.

POLLPRI

高优先级数据(带外)可不阻塞地读取. 这个位使 select 报告在文件上遇到一个异常情况，因为 selct 报告带外数据作为一个异常情况.

POLLHUP

当读这个设备的进程见到文件尾，驱动必须设置 POLLUP(hang-up). 一个调用 select 的进程被告知设备是可读的，如同 selcet 功能所规定的.

POLLERR

一个错误情况已在设备上发生. 当调用 poll，设备被报告位可读可写，因为读写都返回一个错误码而不阻塞.

POLLOUT

这个位在返回值中设置，如果设备可被写入而不阻塞.

POLLWRNORM

这个位和 POLLOUT 有相同的含义，并且有时它确实是相同的数. 一个可写的设备返回(POLLOUT|POLLWRNORM).

POLLWRBAND

如同 POLLRDBAND ，这个位意思是带有零优先级的数据可写入设备. 只有 poll 的数据报实现使用这个位，因为一个数据报看传送带外数据.

应当重复一下 POLLRDBAND 和 POLLWRBAND 仅仅对关联到 socket 的文件描述符有意义: 通常设备驱动不使用这些标志.

poll 的描述使用了大量在实际使用中相对简单的东西. 考虑 poll 方法的 scullpipe 实现:

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
        struct scull_pipe *dev = filp->private_data;
        unsigned int mask = 0;

        /*
        * The buffer is circular; it is considered full
        * if "wp" is right behind "rp" and empty if the
        * two are equal.
        */
        down(&dev->sem);
        poll_wait(filp, &dev->inq,  wait);
        poll_wait(filp, &dev->outq, wait);
        if (dev->rp != dev->wp)
                mask |= POLLIN | POLLRDNORM;  /* readable */
        if (spacefree(dev))
                mask |= POLLOUT | POLLWRNORM;  /* writable */
        up(&dev->sem);
        return mask;
}
```

这个代码简单地增加了 2 个 scullpipe 等待队列到 poll_table，接着设置正确的掩码位，根据数据是否可以读或写.

所示的 poll 代码缺乏文件尾支持，因为 scullpipe 不支持文件尾情况. 对大部分真实的设备，poll 方法应当返回 POLLHUP 如果没有更多数据(或者将)可用. 如果调用者使用 select 系统调用，文件被报告为可读. 不管是使用 poll 还是 select，应用程序知道它能够调用 read 而不必永远等待，并且 read 方法返回 0 来指示文件尾.

例如，对于 真正的 FIFO，读者见到一个文件尾当所有的写者关闭文件，而在 scullpipe 中读者永远见不到文件尾. 这个做法不同是因为 FIFO 是用作一个 2 个进程的通讯通道，而 scullpipe 是一个垃圾桶，人人都可以放数据只要至少有一个读者. 更多地，重新实现内核中已有的东西是没有意义的，因此我们选择在我们的例子里实现一个不同的做法.

与 FIFO 相同的方式实现文件尾就意味着检查 dev->nwwriters，在 read 和 poll 中，并且报告文件尾(如刚刚描述过的)如果没有进程使设备写打开. 不幸的是，使用这个实现方法，如果一个读者打开 scullpipe 设备在写者之前，它可能见到文件尾而没有机会来等待数据. 解决这个问题的最好的方式是在 open 中实现阻塞，如同真正的 FIFO 所做的；这个任务留作读者的一个练习.

6.3.1. 与 read 和 write 的交互

poll 和 select 调用的目的是提前决定是否一个 I/O 操作会阻塞. 在那个方面，它们补充了 read 和 write. 更重要的是，poll 和 select ，因为它们使应用程序同时等待几个数据流，尽管我们在 scull 例子里没有采用这个特性.

一个正确的实现对于使应用程序正确工作是必要的：尽管下列的规则或多或少已经说明过，我们在此总结它们.

6.3.1.1. 从设备中读数据

- 如果在输入缓冲中有数据，read 调用应当立刻返回，没有可注意到的延迟，即便数据少于应用程序要求的，并且驱动确保其他的数据会很快到达. 你可一直返回小于你被请求的数据，如果因为任何理由而方便这样(我们在 scull 中这样做)，如果你至少返回一个字节. 在这个情况下，poll 应当返回 POLLIN|POLLRDNORM.
- 如果在输入缓冲中没有数据，缺省地 read 必须阻塞直到有一个字节. 如果 O_NONBLOCK 被置位，另一方面，read 立刻返回 -EAGIN (尽管一些老版本 SYSTEM V 返回 0 在这个情况时). 在这些情况中，poll 必须报告这个设备是不可读的直到至少一个字节到达. 一旦在缓冲中有数据，我们就回到前面的情况.
- 如果我们处于文件尾，read 应当立刻返回一个 0，不管是否阻塞. 这种情况 poll 应该报告 POLLHUP.

6.3.1.2. 写入设备

- 如果在输出缓冲中有空间，write 应当不延迟返回. 它可接受小于这个调用所请求的数据，但是它必须至少接受一个字节. 在这个情况下，poll 报告这个设备是可写的，通过返回 POLLOUT|POLLWRNORM.
- 如果输出缓冲是满的，缺省地 write 阻塞直到一些空间被释放. 如果 O_NOBLOCK 被设置，write 立刻返回一个 -EAGAIN(老式的 System V Unices 返回 0). 在这些情况下，poll 应当报告文件是不可写的. 另一方面，如果设备不能接受任何多余数据，write 返回 -ENOSPC("设备上没有空间")，不管是否设置了 O_NONBLOCK.
- 在返回之前不要调用 wait 来传送数据，即便当 O_NONBLOCK 被清除. 这是因为许多应用程序选择来找出一个 write 是否会阻塞. 如果设备报告可写，调用必须不阻塞. 如果使用设备的程序想保证它加入到输出缓冲中的数据被真正传送，驱动必须提供一个 fsync 方法. 例如，一个可移除的设备应当有一个 fsnyc 入口.

尽管有一套通用的规则，还应当认识到每个设备是唯一的并且有时这些规则必须稍微弯曲一下. 例如，面向记录的设备(例如磁带设备)无法执行部分写.

6.3.1.3. 刷新挂起的输出

我们已经见到 write 方法如何自己不能解决全部的输出需要. fsync 函数，由同名的系统调用而调用，填补了这个空缺. 这个方法原型是:

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

如果一些应用程序需要被确保数据被发送到设备，fsync 方法必须被实现为不管 O_NONBLOCK 是否被设置. 对 fsync 的调用应当只在设备被完全刷新时返回(即，输出缓冲为空)，即便这需要一些时间. datasync 参数用来区分 fsync 和 fdatasync 系统调用；这样，它只对文件系统代码有用，驱动可以忽略它.

fsync 方法没有不寻常的特性. 这个调用不是时间关键的, 因此每个设备驱动可根据作者的口味实现它. 大部分的时间, 字符驱动只有一个 NULL 指针在它们的 fops 中. 阻塞设备, 另一方面, 常常实现这个方法使用通用的 block_fsync, 它接着会刷新设备的所有的块.

6.3.2. 底层的数据结构

poll 和 select 系统调用的真正实现是相当地简单, 对那些感兴趣于它如何工作的人; epoll 更加复杂一点但是建立在同样的机制上. 无论何时用户应用程序调用 poll, select, 或者 epoll_ctl,[24]24 内核调用这个系统调用所引用的所有文件的 poll 方法, 传递相同的 poll_table 到每个. poll_table 结构只是对一个函数的封装, 这个函数建立了实际的数据结构. 那个数据结构, 对于 poll和 select, 是一个内存页的链表, 其中包含 poll_table_entry 结构. 每个 poll_table_entry 持有被传递给 poll_wait 的 struct file 和 wait_queue_head_t 指针, 以及一个关联的等待队列入口. 对 poll_wait 的调用有时还添加这个进程到给定的等待队列. 整个的结构必须由内核维护以至于这个进程可被从所有的队列中去除, 在 poll 或者 select 返回之前.

如果被轮询的驱动没有一个指示 I/O 可不阻塞地发生, poll 调用简单地睡眠直到一个它所在的等待队列(可能许多)唤醒它.

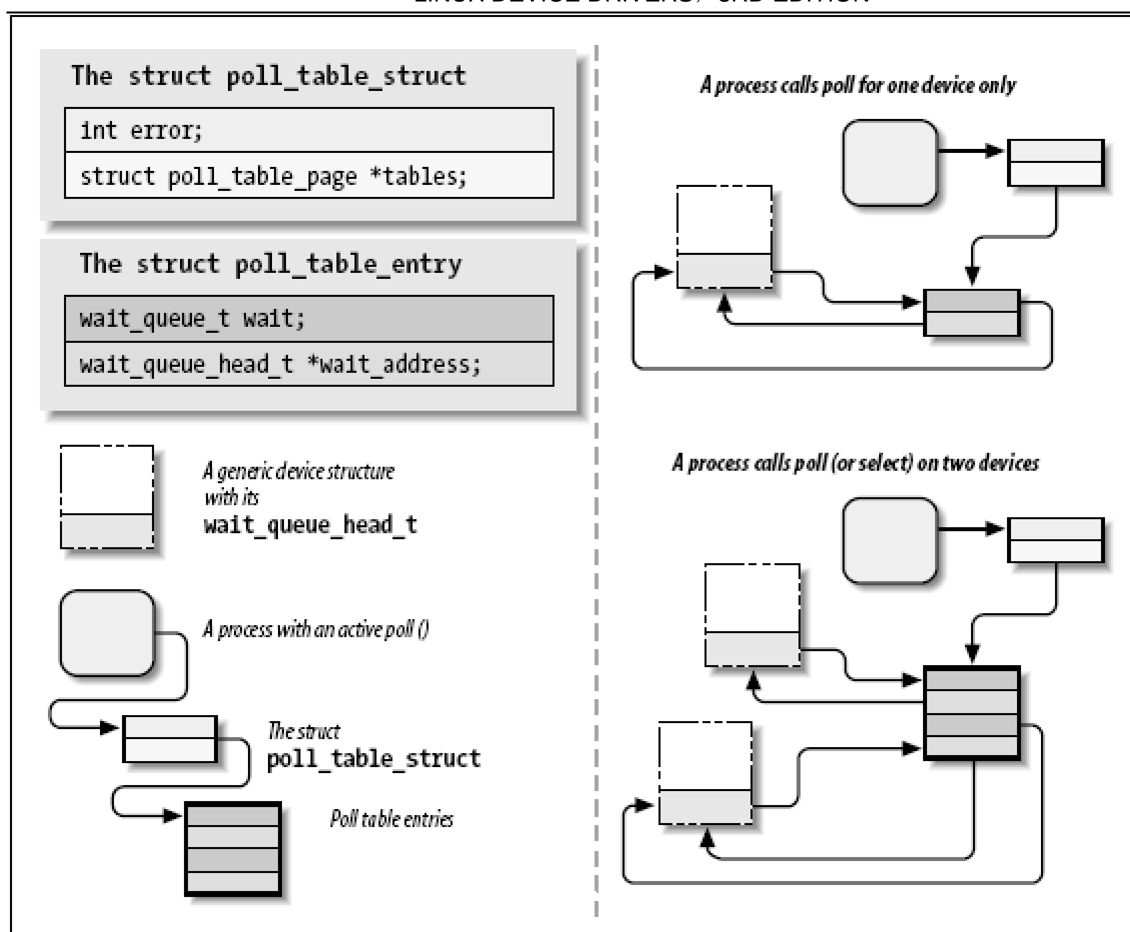
在 poll 实现中有趣的是驱动的 poll 方法可能被用一个 NULL 指针作为 poll_table 参数. 这个情况出现由于几个理由. 如果调用 poll 的应用程序已提供了一个 0 的超时值(指示不应当做等待), 没有理由来堆积等待队列, 并且系统简单地不做它. poll_table 指针还被立刻设置为 NULL 在任何被轮询的驱动指示可以 I/O 之后. 因为内核在那一点知道不会发生等待, 它不建立等待队列链表.

当 poll 调用完成, poll_table 结构被去分配, 并且所有的之前加入到 poll 表的等待队列入口被从表和它们的等待队列中移出.

我们试图在图 poll 背后的数据结构中展示包含在轮询中的数据结构; 这个图是真实数据结构的简化地表示, 因为它忽略了一个 poll 表地多页性质并且忽略了每个 poll_table_entry 的文件指针.

图 6.1. poll 背后的数据结构

24 [24] 这是建立内部数据结构为将来调用 epoll_wait 的函数.



在此，可能理解在新的系统调用 epoll 后面的动机. 在一个典型的情况中，一个对 poll 或者 select 的调用只包括一组文件描述符，所以设置数据结构的开销是小的. 但是，有应用程序在那里，它们使用几千个文件描述符. 在这时，在每次 I/O 操作之间设置和销毁这个数据结构变得非常昂贵. epoll 系统调用家族允许这类应用程序建立内部的内核数据结构只一次，并且多次使用它们.

6.4. 异步通知

尽管阻塞和非阻塞操作和 select 方法的结合对于查询设备在大部分时间是足够的，一些情况还不能被我们迄今所见到的技术来有效地解决.

让我们想象一个进程，在低优先级上执行一个长计算循环，但是需要尽可能快的处理输入数据. 如果这个进程在响应新的来自某些数据获取外设的报告，它应当立刻知道当新数据可用时. 这个应用程序可能被编写来调用 poll 有规律地检查数据，但是，对许多情况，有更好的方法. 通过使能异步通知，这个应用程序可能接受一个信号无论何时数据可用并且不需要让自己去查询.

用户程序必须执行 2 个步骤来使能来自输入文件的异步通知. 首先，它们指定一个进程作为文件的拥有者. 当一个进程使用 fcntl 系统调用发出 F_SETOWN 命令，这个拥有者进程的 ID 被保存在 filp->f_owner 给以后使用. 这一步对内核知道通知谁是必要的. 为了真正使能异步通知，用户程序必须设置 FASYNC 标志在设备中，通过 F_SETFL fcntl 命令.

在这 2 个调用已被执行后，输入文件可请求递交一个 SIGIO 信号，无论何时新数据到达. 信号被发送给存储于 filp->f_owner 中的进程(或者进程组，如果值为负值).

例如，下面的用户程序中的代码行使能了异步的通知到当前进程，给 stdin 输入文件:

```
signal(SIGIO, &input_handler); /* dummy sample; sigaction() is better */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

这个在源码中名为 asynctest 的程序是一个简单的程序，读取 stdin. 它可用来测试 scullpipe 的异步能力. 这个程序和 cat 类似但是不结束于文件尾；它只响应输入，而不是没有输入.

注意，但是，不是所有的设备都支持异步通知，并且你可选择不提供它. 应用程序常常假定异步能力只对 socket 和 tty 可用.

输入通知有一个剩下的问题. 当一个进程收到一个 SIGIO，它不知道哪个输入文件有新数据提供. 如果多于一个文件被使能异步地通知挂起输入的进程，应用程序必须仍然靠 poll 或者 select 来找出发生了什么.

6.4.1. 驱动的观点

对我们来说一个更相关的主题是设备驱动如何实现异步信号. 下面列出了详细的操作顺序，从内核的观点:

- 1. 当发出 F_SETOWN，什么都没发生，除了一个值被赋值给 filp->f_owner.
- 2. 当 F_SETFL 被执行来打开 FASYNC，驱动的 fasync 方法被调用. 这个方法被调用无论何时 FASYNC 的值在 filp->f_flags 中被改变来通知驱动这个变化，因此它可正确地响应. 这个标志在文件被打开时缺省地被清除. 我们将看这个驱动方法的标准实现，在本节.
- 3. 当数据到达，所有的注册异步通知的进程必须被发出一个 SIGIO 信号.

虽然实现第一步是容易的--在驱动部分没有什么要做的--其他的步骤包括维护一个动态数据结构来跟踪不同的异步读者；可能有几个. 这个动态数据结构，但是，不依赖特殊的设备，并且内核提供了一个合适的通用实现这样你不必重新编写同样的代码给每个驱动.

Linux 提供的通用实现是基于一个数据结构和 2 个函数(它们在前面所说的第 2 步和第 3 步被调用). 声明相关材料的头文件是<linux/fs.h>(这里没新东西)，并且数据结构被称为 struct fasync_struct. 至于等待队列，我们需要插入一个指针在设备特定的数据结构中.

驱动调用的 2 个函数对应下面的原型:

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct
**fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

fasync_helper 被调用来从相关的进程列表中添加或去除入口项，当 FASYNC 标志因一个打开文件而改变．它的所有参数除了最后一个，都被提供给 fasync 方法并且被直接传递．当数据到达时 kill_fasync 被用来通知相关的进程．它的参数是被传递的信号(常常是 SIGIO)和 band，这几乎都是 POLL_IN[25] 25(但是这可用来发送"紧急"或者带外数据，在网络代码里)．

这是 scullpipe 如何实现 fasync 方法的:

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
 struct scull_pipe *dev = filp->private_data;
 return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

显然所有的工作都由 fasync_helper 进行．但是，不可能实现这个功能在没有一个方法在驱动里的情况下，因为这个帮忙函数需要存取正确的指向 struct fasync_struct (这里是 与 dev->async_queue)的指针，并且只有驱动可提供这个信息．

当数据到达，下面的语句必须被执行来通知异步读者．因为对 sucllpipe 读者的新数据通过一个发出 write 的进程被产生，这个语句出现在 scullpipe 的 write 方法中．

```
if (dev->async_queue)
 kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

注意，一些设备还实现异步通知来指示当设备可被写入时；在这个情况，当然，kill_fasnyc 必须被使用一个 POLL_OUT 模式来调用．

可能会出现我们已经完成但是仍然有一件事遗漏．我们必须调用我们的 fasync 方法，当文件被关闭来从激活异步读者列表中去除文件．尽管这个调用仅当 filp->f_flags 被设置为 FASYNC 时需要，调用这个函数无论如何不会有问题并且是常见的实现．下面的代码行，例如，是 scullpipe 的 release 方法的一部分:

```
/* remove this filp from the asynchronously notified filp's */
scull_p_fasync(-1, filp, 0);
```

这个在异步通知之下的数据结构一直和结构 struct wait_queue 是一致的，因为 2 种情况都涉及等待一个事件．区别是这个 struct file 被用来替代 struct task_struct. 队列中的结构 file 接着用来存取 f_owner，为了通知进程．

6.5. 移位一个设备

本章最后一个需要我们涉及的东西是 llseek 方法，它有用(对于某些设备)并且容易实现．

25 [25] POLL_IN 是一个符号，用在异步通知代码中；它等同于 POLLIN|POLLRDNORM.

6.5.1. llseek 实现

llseek 方法实现了 lseek 和 llseek 系统调用. 我们已经说了如果 llseek 方法从设备的操作中缺失, 内核中的缺省的实现进行移位通过修改 filp->f_pos, 这是文件中的当前读写位置. 请注意对于 lseek 系统调用要正确工作, 读和写方法必须配合, 通过使用和更新它们收到的作为的参数的 offset 项.

你可能需要提供你自己的方法, 如果移位操作对应一个在设备上的物理操作. 一个简单的例子可在 scull 驱动中找到:

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
        struct scull_dev *dev = filp->private_data;
        loff_t newpos;

        switch(whence)
        {
        case 0: /* SEEK_SET */
                newpos = off;
                break;

        case 1: /* SEEK_CUR */
                newpos = filp->f_pos + off;
                break;

        case 2: /* SEEK_END */
                newpos = dev->size + off;
                break;

        default: /* can't happen */
                return -EINVAL;
        }
        if (newpos < 0)
                return -EINVAL;
        filp->f_pos = newpos;
        return newpos;
}
```

唯一设备特定的操作是从设备中获取文件长度. 在 scull 中 read 和 write 方法如需要地一样协作, 如同在第 3 章所示.

尽管刚刚展示的这个实现对 scull 有意义, 它处理一个被很好定义了的数据区, 大部分设备提供了一个数据流而不是一个数据区(想想串口或者键盘), 并且移位这些设备没有意义. 如果这就是你的设备的情况, 你不能只制止声明 llseek 操作, 因为缺省的方法允许移位. 相反, 你应当通知内核你的设备不支持 llseek , 通过调用 nonseekable_open 在你的 open 方法中.

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

这个调用标识了给定的 filp 为不可移位的；内核从不允许一个 lseek 调用在这样一个文件上成功. 通过用这样的方式标识这个文件, 你可确定不会有通过 pread 和 pwrite 系统调用的方式来试图移位这个文件.

完整起见, 你也应该在你的 file_operations 结构中设置 llseek 方法到一个特殊的帮忙函数 no_llseek, 它定义在 <linux/fs.h>.

6.6. 在一个设备文件上的存取控制

提供存取控制对于一个设备节点来说有时是至关重要的. 不仅是非授权用户不能使用设备(由文件系统许可位所强加的限制), 而且有时只有授权用户才应当被允许来打开设备一次.

这个问题类似于使用 ttys 的问题. 在那个情况下, login 进程改变设备节点的所有权, 无论何时一个用户登录到系统, 为了阻止其他的用户打扰或者偷听这个 tty 的数据流. 但是, 仅仅为了保证对它的唯一读写而使用一个特权程序在每次打开它时改变一个设备的拥有权是不实际的.

迄今所显示的代码没有实现任何的存取控制, 除了文件系统许可位. 如果系统调用 open 将请求递交给驱动, open 就成功了. 我们现在介绍几个新技术来实现一些额外的检查.

每个在本节中展示的设备有和空的 scull 设备有相同的行为(即, 它实现一个持久的内存区)但是在存取控制方面和 scull 不同, 这个实现在 open 和 release 操作中.

6.6.1. 单 open 设备

提供存取控制的强力方式是只允许一个设备一次被一个进程打开(单次打开). 这个技术最好是避免因为它限制了用户的灵活性. 一个用户可能想运行不同的进程在一个设备上, 一个读状态信息而另一个写数据. 在某些情况下, 用户通过一个外壳脚本运行几个简单的程序可做很多事情, 只要它们可并发存取设备. 换句话说, 实现一个单 open 行为实际是在创建策略, 这样可能会介入你的用户要做的范围.

只允许单个进程打开设备有不期望的特性, 但是它也是一个设备驱动最简单实现的存取控制, 因此它在这里被展示. 这个源码是从一个称为 scullsingle 的设备中提取的.

scullsingle 设备维护一个 atiomic_t 变量, 称为 scull_s_available; 这个变量被初始化为值 1, 表示设备确实可用. open 调用递减并测试 scull_s_available 并拒绝存取如果其他人已经使设备打开.

```
static atomic_t scull_s_available = ATOMIC_INIT(1);
static int scull_s_open(struct inode *inode, struct file *filp)
{

        struct scull_dev *dev = &scull_s_device; /* device information */
        if (! atomic_dec_and_test (&scull_s_available))
        {
```

```
                atomic_inc(&scull_s_available);
                return -EBUSY; /* already open */
        }

        /* then, everything else is copied from the bare scull device */
        if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)

                scull_trim(dev);
        filp->private_data = dev;
        return 0; /* success */
}
```

release 调用，另一方面，标识设备为不再忙:

```
static int scull_s_release(struct inode *inode, struct file *filp)
{
        atomic_inc(&scull_s_available); /* release the device */
        return 0;
}
```

正常地，我们建议你将 open 标志 scul_s_available 放在设备结构中(scull_dev 这里)，因为，从概念上，它属于这个设备. scull 驱动，但是，使用独立的变量来保持这个标志，因此它可使用和空 scull 设备同样的设备结构和方法，并且最少的代码复制.

6.6.2. 一次对一个用户限制存取

单打开设备之外的下一步是使一个用户在多个进程中打开一个设备，但是一次只允许一个用户打开设备. 这个解决方案使得容易测试设备，因为用户一次可从几个进程读写，但是假定这个用户负责维护在多次存取中的数据完整性. 这通过在 open 方法中添加检查来实现；这样的检查在通常的许可检查后进行，并且只能使存取更加严格，比由拥有者和组许可位所指定的限制. 这是和 ttys 所用的存取策略是相同的，但是它不依赖于外部的特权程序.

这些存取策略实现地有些比单打开策略要奇怪. 在这个情况下，需要 2 项：一个打开计数和设备拥有者 uid. 再一次，给这个项的最好的地方是在设备结构中；我们的例子使用全局变量代替，是因为之前为 scullsingle 所解释的的原因. 这个设备的名子是 sculluid.

open 调用在第一次打开时同意了存取但是记住了设备拥有者. 这意味着一个用户可打开设备多次，因此允许协调多个进程对设备并发操作. 同时，没有其他用户可打开它，这样避免了外部干扰. 因为这个函数版本几乎和之前的一致，这样相关的部分在这里被复制:

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
                (scull_u_owner != current->uid) && /* allow user */
                (scull_u_owner != current->euid) && /* allow whoever did su */
                !capable(CAP_DAC_OVERRIDE))
```

```
{ /* still allow root */
        spin_unlock(&scull_u_lock);
        return -EBUSY; /* -EPERM would confuse the user */
}

if (scull_u_count == 0)
        scull_u_owner = current->uid; /* grab it */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

注意 sculluid 代码有 2 个变量 (scull_u_owner 和 scull_u_count)来控制对设备的存取，并且这样可被多个进程并发地存取. 为使这些变量安全，我们使用一个自旋锁控制对它们的存取(scull_u_lock). 没有这个锁，2 个(或多个)进程可同时测试 scull_u_count ，并且都可能认为它们拥有设备的拥有权. 这里使用一个自旋锁，是因为这个锁被持有极短的时间，并且驱动在持有这个锁时不做任何可睡眠的事情.

我们选择返回 -EBUSY 而不是 -EPERM，即便这个代码在进行许可检测，为了给一个被拒绝存取的用户指出正确的方向. 对于"许可拒绝"的反应常常是检查 /dev 文件的模式和拥有者，而"设备忙"正确地建议用户应当寻找一个已经在使用设备的进程.

这个代码也检查来看是否正在试图打开的进程有能力来覆盖文件存取许可；如果是这样，open 被允许即便打开进程不是设备的拥有者. CAP_DAC_OVERRIDE 能力在这个情况中适合这个任务.

release 方法看来如下:

```
static int scull_u_release(struct inode *inode, struct file *filp)
{
        spin_lock(&scull_u_lock);
        scull_u_count--; /* nothing else */
        spin_unlock(&scull_u_lock);
        return 0;
}
```

再次，我们在修改计数之前必须获得锁，来确保我们没有和另一个进程竞争.

6.6.3. 阻塞 open 作为对 EBUSY 的替代

当设备不可存取，返回一个错误常常是最合理的方法，但是有些情况用户可能更愿意等待设备.

例如，如果一个数据通讯通道既用于规律地预期地传送报告(使用 crontab)，也用于根据用户的需要偶尔地使用，对于被安排的操作最好是稍微延迟，而不是只是因为通道当前忙而失败.

这是程序员在设计一个设备驱动时必须做的一个选择之一，并且正确的答案依赖正被解决的实际问题.

对 EBUSY 的替代，如同你可能已经想到的，是实现阻塞 open. scullwuid 设备是一个在打开时等待设备而不是返回 -EBUSY 的 sculluid 版本. 它不同于 sculluid 只在下面的打开操作部分:

```
spin_lock(&scull_w_lock);
while (! scull_w_available())
{
        spin_unlock(&scull_w_lock);
        if (filp->f_flags & O_NONBLOCK)
                return -EAGAIN;
        if (wait_event_interruptible (scull_w_wait, scull_w_available()))
                return -ERESTARTSYS; /* tell the fs layer to handle it */
        spin_lock(&scull_w_lock);
}
if (scull_w_count == 0)
        scull_w_owner = current->uid; /* grab it */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

这个实现再次基于一个等待队列. 如果设备当前不可用，试图打开它的进程被放置到等待队列直到拥有进程关闭设备.

release 方法，接着，负责唤醒任何挂起的进程:

```
static int scull_w_release(struct inode *inode, struct file *filp)
{

 int temp;
 spin_lock(&scull_w_lock);
 scull_w_count--;
 temp = scull_w_count;
 spin_unlock(&scull_w_lock);
    if (temp == 0)
 wake_up_interruptible_sync(&scull_w_wait); /* awake other uid's */
 return 0;
}
```

这是一个例子，这里调用 wake_up_interruptible_sync 是有意义的. 当我们做这个唤醒，我们只是要返回到用户空间，这对于系统是一个自然的调度点. 当我们做这个唤醒时不是潜在地重新调度，最好只是调用 "sync" 版本并且完成我们的工作.

阻塞式打开实现的问题是对于交互式用户真的不好，他们不得不猜想哪里出错了. 交互式用户常常调用标准命令，例如 cp 和 tar，并且不能增加 O_NONBLOCK 到 open 调用. 有

些使用磁带驱动器做备份的人可能喜欢有一个简单的"设备或者资源忙"消息，来替代被扔在一边猜为什么今天的硬盘驱动器这么安静，此时 tar 应当在扫描它.

这类的问题(需要一个不同的，不兼容的策略对于同一个设备)最好通过为每个存取策略实现一个设备节点来实现. 这个做法的一个例子可在 linux 磁带驱动中找到，它提供了多个设备文件给同一个设备. 例如，不同的设备文件将使驱动器使用或者不用压缩记录，或者自动回绕磁带当设备被关闭时.

6.6.4. 在 open 时复制设备

管理存取控制的另一个技术是创建设备的不同的私有拷贝，根据打开它的进程.

明显地，这只当设备没有绑定到一个硬件实体时有可能；scull 是一个这样的"软件"设备的例子. /dev/tty 的内部使用类似的技术来给它的进程一个不同的 /dev 入口点呈现的视图. 当设备的拷贝被软件驱动创建，我们称它们为虚拟设备--就象虚拟控制台使用一个物理 tty 设备.

结构这类的存取控制很少需要，这个实现可说明内核代码是多么容易改变应用程序的对周围世界的看法(即，计算机).

/dev/scullpriv 设备节点在 scull 软件包只实现虚拟设备. scullpriv 实现使用了进程的控制 tty 的设备号作为对存取虚拟设备的钥匙. 但是，你可以轻易地改变代码来使用任何整数值作为钥匙；每个选择都导致一个不同的策略. 例如，使用 uid 导致一个不同地虚拟设备给每个用户，而使用一个 pid 钥匙创建一个新设备为每个存取它的进程.

使用控制终端的决定打算用在易于使用 I/O 重定向测试设备：设备被所有的在同一个虚拟终端运行的命令所共享，并且保持独立于在另一个终端上运行的命令所见到的.

open 方法看来象下面的代码. 它必须寻找正确的虚拟设备并且可能创建一个. 这个函数的最后部分没有展示，因为它拷贝自空的 scull，我们已经见到过.

```
/* The clone-specific data structure includes a key field */
struct scull_listitem
{
        struct scull_dev device;
        dev_t key;
        struct list_head list;

};
/* The list of devices, and a lock to protect it */
static LIST_HEAD(scull_c_list);
static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;

/* Look for a device or create one if missing */
static struct scull_dev *scull_c_lookfor_device(dev_t key)
{
```

```
        struct scull_listitem *lptr;
        list_for_each_entry(lptr, &scull_c_list, list)
        {
                if (lptr->key == key)
                        return &(lptr->device);
        }

        /* not found */
        lptr = kmalloc(sizeof(struct scull_listitem), GFP_KERNEL);
        if (!lptr)
                return NULL;
        /* initialize the device */
        memset(lptr, 0, sizeof(struct scull_listitem));
        lptr->key = key;
        scull_trim(&(lptr->device)); /* initialize it */
        init_MUTEX(&(lptr->device.sem));

        /* place it in the list */
        list_add(&lptr->list, &scull_c_list);

        return &(lptr->device);
}

static int scull_c_open(struct inode *inode, struct file *filp)
{
        struct scull_dev *dev;

        dev_t key;
        if (!current->signal->tty)
        {
                PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
                return -EINVAL;

        }
        key = tty_devnum(current->signal->tty);

        /* look for a scullc device in the list */
        spin_lock(&scull_c_lock);
        dev = scull_c_lookfor_device(key);
        spin_unlock(&scull_c_lock);

        if (!dev)
                return -ENOMEM;

        /* then, everything else is copied from the bare scull device */
```

这个 release 方法没有做特殊的事情. 它将在最后的关闭时正常地释放设备, 但是我们不选择来维护一个 open 计数而来简化对驱动的测试. 如果设备在最后的关闭被释放, 你将不能读相同的数据在写入设备之后, 除非一个后台进程将保持它打开. 例子驱动采用了简单的方法来保持数据, 以便在下一次打开时, 你会发现它在那里. 设备在 scull_cleanup 被调用时释放.

这个代码使用通用的 linux 链表机制, 而不是从头开始实现相同的功能. linux 链表在第 11 章中讨论.

这里是 /dev/scullpriv 的 release 实现, 它结束了对设备方法的讨论.

```
static int scull_c_release(struct inode *inode, struct file *filp)
{
        /*
        *Nothing to do, because the device is persistent.
        *A `real' cloned device should be freed on last close */

        return 0;
}
```

6.7. 快速参考

本章介绍了下面的符号和头文件:

#include <linux/ioctl.h>

声明用来定义 ioctl 命令的宏定义. 当前被 <linux/fs.h> 包含.

_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS

ioctl 命令的不同位段所使用的位数. 还有 4 个宏来指定 MASK 和 4 个指定 SHIFT, 但是它们主要是给内部使用. _IOC_SIZEBIT 是一个要检查的重要的值, 因为它跨体系改变.

_IOC_NONE
_IOC_READ
_IOC_WRITE

"方向"位段可能的值. "read" 和 "write" 是不同的位并且可相或来指定 read/write. 这些值是基于 0 的.

_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)

```
_IOW(type,nr,size)
_IOWR(type,nr,size)
```

用来创建 ioclt 命令的宏定义.

```
_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)
```

用来解码一个命令的宏定义. 特别地, _IOC_TYPE(nr) 是 _IOC_READ 和 _IOC_WRITE 的 OR 结合.

```
#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);
```

检查一个用户空间的指针是可用的. access_ok 返回一个非零值, 如果应当允许存取.

```
VERIFY_READ
VERIFY_WRITE
```

access_ok 中 type 参数的可能取值. VERIFY_WRITE 是 VERIFY_READ 的超集.

```
#include <asm/uaccess.h>
int put_user(datum,ptr);
int get_user(local,ptr);
int __put_user(datum,ptr);
int __get_user(local,ptr);
```

用来存储或获取一个数据到或从用户空间的宏. 传送的字节数依赖 sizeof(*ptr). 常规的版本调用 access_ok , 而常规版本(__put_user 和 __get_user) 假定 access_ok 已经被调用了.

```
#include <linux/capability.h>
```

定义各种 CAP_ 符号, 描述一个用户空间进程可有的能力.

```
int capable(int capability);
```

返回非零值如果进程有给定的能力.

```
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);
```

Linux 等待队列的定义类型. 一个 wait_queue_head_t 必须被明确在运行时使用 init_waitqueue_head 或者编译时使用 DEVLARE_WAIT_QUEUE_HEAD 进行初始化.

```
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int condition);
int wait_event_timeout(wait_queue_head_t q, int condition, int time);
int wait_event_interruptible_timeout(wait_queue_head_t q, int condition,int
time);
```

使进程在给定队列上睡眠, 直到给定条件值为真值.

```
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_nr(struct wait_queue **q, int nr);
void wake_up_interruptible_nr(struct wait_queue **q, int nr);
void wake_up_all(struct wait_queue **q);
void wake_up_interruptible_all(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

唤醒在队列 q 上睡眠的进程. _interruptible 的形式只唤醒可中断的进程. 正常地, 只有一个互斥等待者被唤醒, 但是这个行为可被 _nr 或者 _all 形式所改变. _sync 版本在返回之前不重新调度 CPU.

```
#include <linux/sched.h>
set_current_state(int state);
```

设置当前进程的执行状态. TASK_RUNNING 意味着它已经运行, 而睡眠状态是 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE.

```
void schedule(void);
```

选择一个可运行的进程从运行队列中. 被选中的进程可是当前进程或者另外一个.

```
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);
```

wait_queue_t 类型用来放置一个进程到一个等待队列.

```
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait,
int state);
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

帮忙函数, 可用来编码一个手工睡眠.

```
void sleep_on(wiat_queue_head_t *queue);
void interruptible_sleep_on(wiat_queue_head_t *queue);
```

老式的不推荐的函数，它们无条件地使当前进程睡眠.

```
#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p);
```

将当前进程放入一个等待队列，不立刻调度. 它被设计来被设备驱动的 poll 方法使用.

```
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);
```

一个"帮忙者"，来实现 fasync 设备方法. mode 参数是传递给方法的相同的值，而 fa 指针指向一个设备特定的 fasync_struct *.

```
void kill_fasync(struct fasync_struct *fa, int sig, int band);
```

如果这个驱动支持异步通知，这个函数可用来发送一个信号到登记在 fa 中的进程.

```
int nonseekable_open(struct inode *inode, struct file *filp);
loff_t no_llseek(struct file *file, loff_t offset, int whence);
```

nonseekable_open 应当在任何不支持移位的设备的 open 方法中被调用. 这样的设备应当使用 no_llseek 作为它们的 llseek 方法.

第 7 章 时间，延时，和延后工作

到此，我们知道了如何编写一个全特性字符模块的基本知识．真实世界的驱动，然而，需要做比实现控制一个设备的操作更多的事情；它们不得不处理诸如定时，内存管理，硬件存取，等更多．幸运的是，内核输出了许多设施来简化驱动编写者的任务．在下几章中，我们将描述一些你可使用的内核资源．这一章引路，通过描述定时问题是如何阐述．处理时间问题包括下列任务，按照复杂度上升的顺序：

- 测量时间流失和比较时间
- 知道当前时间
- 指定时间量的延时操作
- 调度异步函数在之后的时间发生

7.1. 测量时间流失

内核通过定时器中断来跟踪时间的流动．中断在第 10 章详细描述．

定时器中断由系统定时硬件以规律地间隔产生；这个间隔在启动时由内核根据 HZ 值来编程，HZ 是一个体系依赖的值，在 <linux/param.h>中定义或者它所包含的一个子平台文件中．在发布的内核源码中的缺省值在真实硬件上从 50 到 1200 嘀哒每秒，在软件模拟器中往下到 24．大部分平台运行在 100 或者 1000 中断每秒；流行的 x86 PC 缺省是 1000，尽管它在以前版本上(向上直到并且包括 2.4)常常是 100．作为一个通用的规则，即便如果你知道 HZ 的值，在编程时你应当从不依赖这个特定值．

可能改变 HZ 的值，对于那些要系统有一个不同的时钟中断频率的人．如果你在头文件中改变 HZ 的值，你需要使用新的值重编译内核和所有的模块．如果你愿意付出额外的时间中断的代价来获得你的目标，你可能想提升 HZ 来得到你的异步任务的更细粒度的精度．实际上，提升 HZ 到 1000 在使用 2.4 或 2.2 内核版本的 x86 工业系统中是相当普遍的．但是，对于当前版本，最好的方法是保持 HZ 的缺省值，由于我们完全信任内核开发者，他们肯定已经选择了最好的值．另外，一些内部计算当前实现为只为从 12 到 1535 范围的 HZ (见 <linux/timex.h> 和 RFC-1589)．

每次发生一个时钟中断，一个内核计数器的值递增．这个计数器在系统启动时初始化为 0，因此它代表从最后一次启动以来的时钟嘀哒的数目．这个计数器是一个 64-位 变量(即便在 32-位的体系上)并且称为 jiffies_64．但是，驱动编写者正常地存取 jiffies 变量，一个 unsigned long，或者和 jiffies_64 是同一个或者它的低有效位．使用 jiffies 常常是首选，因为它更快，并且再所有的体系上存取 64-位的 jiffies_64 值不必要是原子的．

除了低精度的内核管理的 jiffy 机制，一些 CPU 平台特有一个高精度的软件可读的计数器．尽管它的实际使用有些在各个平台不同，它有时是一个非常有力的工具．

7.1.1. 使用 jiffies 计数器

这个计数器和来读取它的实用函数位于 <linux/jiffies.h>, 尽管你会常常只是包含 <linux/sched.h>, 它会自动地将 jiffies.h 拉进来. 不用说, jiffies 和 jiffies_64 必须当作只读的.

无论何时你的代码需要记住当前的 jiffies 值, 可以简单地存取这个 unsigned long 变量, 它被声明做 volatile 来告知编译器不要优化内存读. 你需要读取当前的计数器, 无论何时你的代码需要计算一个将来的时间戳, 如下面例子所示:

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n * HZ / 1000; /* n milliseconds */
```

这个代码对于 jiffies 回绕没有问题, 只要不同的值以正确的方式进行比较. 尽管在 32-位 平台上当 HZ 是 1000 时, 计数器只是每 50 天回绕一次, 你的代码应当准备面对这个事件. 为比较你的被缓存的值(象上面的 stamp_1) 和当前值, 你应当使用下面一个宏定义:

```
#include <linux/jiffies.h>
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

第一个当 a, 作为一个 jiffies 的快照, 代表 b 之后的一个时间时, 取值为真, 第二个当 时间 a 在时间 b 之前时取值为真, 以及最后 2 个比较"之后或相同"和"之前或相同". 这个代码工作通过转换这个值为 signed long, 减它们, 并且比较结果. 如果你需要以一种安全的方式知道 2 个 jiffies 实例之间的差, 你可以使用同样的技巧: diff = (long)t2 - (long)t1;.

你可以转换一个 jiffies 差为毫秒, 一般地通过:

```
msec = diff * 1000 / HZ;
```

有时, 但是, 你需要与用户空间程序交换时间表示, 它们打算使用 struct timeval 和 struct timespec 来表示时间. 这 2 个结构代表一个精确的时间量, 使用 2 个成员: seconds 和 microseconds 在旧的流行的 struct timeval 中使用, seconds 和 nanoseconds 在新的 struct timespec 中使用. 内核输出 4 个帮助函数来转换以 jiffies 表达的时间值, 到和从这些结构:

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
```

```
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

存取这个 64-位 jiffy 计数值不象存取 jiffies 那样直接. 而在 64-位 计算机体系上, 这 2 个变量实际上是一个, 存取这个值对于 32-位 处理器不是原子的. 这意味着你可能读到错误的值如果这个变量的两半在你正在读取它们时被更新. 极不可能你会需要读取这个 64-位 计数器, 但是万一你需要, 你会高兴地得知内核输出了一个特别地帮助函数, 为你完成正确地加锁:

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

在上面的原型中, 使用了 u64 类型. 这是一个定义在 <linux/types.h> 中的类型, 在 11 章中讨论, 并且表示一个 unsigned 64-位 类型.

如果你在奇怪 32-位 平台如何同时更新 32-位 和 64-位 计数器, 读你的平台的连接脚本(查找一个文件, 它的名子匹配 valinux*.lds*). 在那里, jiffies 符号被定义来存取这个 64-位 值的低有效字, 根据平台是小端或者大端. 实际上, 同样的技巧也用在 64-位 平台上, 因此这个 unsigned long 和 u64 变量在同一个地址被存取.

最后, 注意实际的时钟频率几乎完全对用户空间隐藏. 宏 HZ 一直扩展为 100 当用户空间程序包含 param.h, 并且每个报告给用户空间的计数器都对应地被转换. 这应用于 clock(3), times(2), 以及任何相关的函数. 对 HZ 值的用户可用的唯一证据是时钟中断多快发生, 如在 /proc/interrupts 所显示的. 例如, 你可以获得 HZ, 通过用在 /proc/uptime 中报告的系统 uptime 除这个计数值.

7.1.2. 处理器特定的寄存器

如果你需要测量非常短时间间隔, 或者你需要非常高精度, 你可以借助平台依赖的资源, 一个要精度不要移植性的选择.

在现代处理器中, 对于经验性能数字的迫切需求被大部分 CPU 设计中内在的指令定时不确定性所阻碍, 这是由于缓存内存, 指令调度, 以及分支预测引起. 作为回应, CPU 制造商引入一个方法来计数时钟周期, 作为一个容易并且可靠的方法来测量时间流失. 因此, 大部分现代处理器包含一个计数器寄存器, 它在每个时钟周期固定地递增一次. 现在, 资格时钟计数器是唯一可靠的方法来进行高精度的时间管理任务.

细节每个平台不同: 这个寄存器可以或者不可以从用户空间可读, 它可以或者不可以写, 并且它可能是 64 或者 32 位宽. 在后一种情况, 你必须准备处理溢出, 就象我们处理 jiffy 计数器一样. 这个寄存器甚至可能对你的平台来说不存在, 或者它可能被硬件设计者在一个外部设备实现, 如果 CPU 缺少这个特性并且你在使用一个特殊用途的计算机.

无论是否寄存器可以被清零, 我们强烈不鼓励复位它, 即便当硬件允许时. 毕竟, 在任何给定时间你可能不是这个计数器的唯一用户; 在一些支持 SMP 的平台上, 例如, 内核依赖这样一个计数器来在处理器之间同步. 因为你可以一直测量各个值的差, 只要差没有超过溢出时间, 你可以通过修改它的当前值来做这个事情不用声明独自拥有这个寄存器.

最有名的计数器寄存器是 TSC (timestamp counter), 在 x86 处理器中随 Pentium 引入的并且在所有从那之后的 CPU 中出现 -- 包括 x86_64 平台. 它是一个 64-位 寄存器计数 CPU 的时钟周期; 它可从内核和用户空间读取.

在包含了 <asm/msr.h> (一个 x86-特定的头文件, 它的名子代表"machine-specific registers"), 你可使用一个这些宏:

```
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var64);
```

第一个宏自动读取 64-位 值到 2 个 32-位 变量; 下一个("read low half") 读取寄存器的低半部到一个 32-位 变量, 丢弃高半部; 最后一个读 64-位 值到一个 long long 变量, 由此得名. 所有这些宏存储数值到它们的参数中.

对大部分的 TSC 应用, 读取这个计数器的的低半部足够了. 一个 1-GHz 的 CPU 只在每 4.2 秒溢出一次, 因此你不会需要处理多寄存器变量, 如果你在使用的时间流失确定地使用更少时间. 但是, 随着 CPU 频率不断上升以及定时需求的提高, 将来你会几乎可能需要常常读取 64-位 计数器.

作为一个只使用寄存器低半部的例子, 下面的代码行测量了指令自身的执行:

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

一些其他的平台提供相似的功能, 并且内核头文件提供一个体系独立的功能, 你可用来代替 rdtsc. 它称为 get_cycles, 定义在 <asm/timex.h>(由 <linux/timex.h> 包含). 它的原型是:

```
 #include <linux/timex.h>
 cycles_t get_cycles(void);
```

这个函数为每个平台定义, 并且它一直返回 0 在没有周期-计数器寄存器的平台上. cycles_t 类型是一个合适的 unsigned 类型来持有读到的值.

不论一个体系独立的函数是否可用, 我们最好利用机会来展示一个内联汇编代码的例子. 为此, 我们实现一个 rdtscl 函数给 MIPS 处理器, 它与在 x86 上同样的方式工作.

拖尾的 nop 指令被要求来阻止编译器在 mfc0 之后马上存取指令中的目标寄存器. 这种内部锁在 RISC 处理器中是典型的, 并且编译器仍然可以在延迟时隙中调度有用的指令. 在这个情况中, 我们使用 nop 因为内联汇编对编译器是一个黑盒并且不会进行优化.[26] [26]

```
#define rdtscl(dest) \
```

[26] [26] 我们在 MIPS 上建立这例子, 因为大部分的 MIPS 处理器特有一个 32-位 计数器作为它们内部"协处理器 0"的寄存器 9. 为存取这个寄存器, 仅仅从内核空间可读, 你可以定义下列的宏来执行一条"从协处理器 0 转移"的汇编指令.

```
 __asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

有这个宏在，MIPS 处理器可以执行同样的代码，如同前面为 x86 展示的一样的代码.

使用 gcc 内联汇编，通用寄存器的分配留给编译器. 刚刚展示的这个宏使用 %0 作为"参数 0"的一个占位符，之后它被指定为"任何用作输出(=)的寄存器(r)". 这个宏还声明输出寄存器必须对应 C 表达式 dest. 内联函数的语法是非常强大但是有些复杂，特别对于那些有限制每个寄存器可以做什么的体系上(就是说，x86 家族). 这个用法在 gcc 文档中描述，常常在 info 文档目录树中有.

本节已展示的这个简短的 C-代码片段已在一个 K7-级 x86 处理器 和一个 MIPS VR4181 (使用刚刚描述过的宏)上运行. 前者报告了一个 11 个时钟嘀哒的时间流失而后者只是 2 个时钟嘀哒. 小的数字是期望的，因为 RISC 处理器常常每个时钟周期执行一条指令.

有另一个关于时戳计数器的事情值得知道：它们在一个 SMP 系统中不必要跨处理器同步. 为保证得到一个一致的值，你应当为查询这个计数器的代码禁止抢占.

7.2. 获知当前时间

内核代码能一直获取一个当前时间的表示，通过查看 jifies 的值. 常常地，这个值只代表从最后一次启动以来的时间，这个事实对驱动来说无关，因为它的生命周期受限于系统的 uptime. 如所示，驱动可以使用 jiffies 的当前值来计算事件之间的时间间隔(例如，在输入驱动中从单击中区分双击或者计算超时). 简单地讲，查看 jiffies 几乎一直是足够的，当你需要测量时间间隔. 如果你需要对短时间流失的非常精确的测量，处理器特定的寄存器来帮忙了(尽管它们带来严重的移植性问题).

它是非常不可能一个驱动会需要知道墙上时钟时间，以月，天，和小时来表达的；这个信息常常只对用户程序需要，例如 cron 和 syslogd. 处理真实世界的时间常常最好留给用户空间，那里的 C 库提供了更好的支持；另外，这样的代码常常太策略相关以至于不属于内核. 有一个内核函数转变一个墙上时钟时间到一个 jiffies 值，但是：

```
#include <linux/time.h>
unsigned long mktime (unsigned int year, unsigned int mon,
 unsigned int day, unsigned int hour,
 unsigned int min, unsigned int sec);
```

重复:直接在驱动中处理墙上时钟时间往往是一个在实现策略的信号，并且应当因此而被置疑.

虽然你不会一定处理人可读的时间表示，有时你需要甚至在内核空间中处理绝对时间. 为此，<linux/time.h> 输出了 do_gettimeofday 函数. 当被调用时，它填充一个 struct timeval 指针 -- 和在 gettimeofday 系统调用中使用的相同 -- 使用类似的秒和毫秒值. do_gettimeofday 的原型是:

```
 #include <linux/time.h>
 void do_gettimeofday(struct timeval *tv);
```

这段源代码声明 do_gettimeofday 有″接近毫秒的精度″，因为它询问时间硬件当前 jiffy 多大比例已经流失. 这个精度每个体系都不同，但是，因为它依赖实际使用中的硬件机制. 例如，一些 m68knommu 处理器，Sun3 系统，和其他 m68k 系统不能提供大于 jiffy 的精度. Pentium 系统，另一方面，提供了非常快速和精确的小于嘀哒的测量，通过读取本章前面描述的时戳计数器.

当前时间也可用(尽管使用 jiffy 的粒度)来自 xtime 变量，一个 struct timespec 值. 不鼓励这个变量的直接使用，因为难以原子地同时存取这 2 个字段. 因此，内核提供了实用函数 current_kernel_time:

```
#include <linux/time.h>
struct timespec current_kernel_time(void);
```

用来以各种方式获取当前时间的代码，可以从由 O' Reilly 提供的 FTP 网站上的源码文件的 jit (″just in time″) 模块获得. jit 创建了一个文件称为 /proc/currentime，当读取时，它以 ASCII 码返回下列项:

- 当前的 jiffies 和 jiffies_64 值，以 16 进制数的形式.
- 如同 do_gettimeofday 返回的相同的当前时间.
- 由 current_kernel_time 返回的 timespec.

我们选择使用一个动态的 /proc 文件来保持样板代码为最小 -- 它不值得创建一整个设备只是返回一点儿文本信息.

这个文件连续返回文本行只要这个模块加载着；每次 read 系统调用收集和返回一套数据，为更好阅读而组织为 2 行. 无论何时你在少于一个时钟嘀哒内读多个数据集，你将看到 do_gettimeofday 之间的差别，它询问硬件，并且其他值仅在时钟嘀哒时被更新.

```
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
 1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
 1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
 1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
 1062370899.630161336
```

在上面的屏幕快照中，由 2 件有趣的事情要注意. 首先，这个 current_kernel_time 值，尽管以纳秒来表示，只有时钟嘀哒的粒度；do_gettimeofday 持续报告一个稍晚的时间但是不晚于下一个时钟嘀哒. 第二，这个 64-位的 jiffies 计数器有 高 32-位字集合的最低有效位. 这是由于 INITIAL_JIFFIES 的缺省值，在启动时间用来初始化计数器，在启动时间后几分钟内强加一个低字溢出来帮助探测与这个刚好溢出相关的问题. 这个在计数器中的初始化偏好没有效果，因为 jiffies 与墙上时钟时间无关. 在 /proc/uptime 中，这里内核从计数器中抽取 uptime，初始化偏好在转换前被去除.

7.3. 延后执行

设备驱动常常需要延后一段时间执行一个特定片段的代码，常常允许硬件完成某个任务. 在这一节我们涉及许多不同的技术来获得延后. 每种情况的环境决定了使用哪种技术最好; 我们全都仔细检查它们，并且指出每一个的长处和缺点.

一件要考虑的重要的事情是你需要的延时如何与时钟嘀哒比较，考虑到 HZ 的跨各种平台的范围. 那种可靠地比时钟嘀哒长并且不会受损于它的粗粒度的延时，可以利用系统时钟. 每个短延时典型地必须使用软件循环来实现. 在这 2 种情况中存在一个灰色地带. 在本章，我们使用短语" long " 延时来指一个多 jiffy 延时，在一些平台上它可以如同几个毫秒一样少，但是在 CPU 和内核看来仍然是长的.

下面的几节讨论不同的延时，通过采用一些长路径，从各种直觉上不适合的方法到正确的方法. 我们选择这个途径因为它允许对内核相关定时方面的更深入的讨论. 如果你急于找出正确的代码，只要快速浏览本节.

7.3.1. 长延时

偶尔地，一个驱动需要延后执行相对长时间 -- 多于一个时钟嘀哒. 有几个方法实现这类延时; 我们从最简单的技术开始，接着进入到高级些的技术.

7.3.1.1. 忙等待

如果你想延时执行多个时钟嘀哒，允许在值中某些疏忽，最容易的(尽管不推荐) 的实现是一个监视 jiffy 计数器的循环. 这种忙等待实现常常看来象下面的代码，这里 j1 是 jiffies 的在延时超时的值:

```
while (time_before(jiffies, j1))
    cpu_relax();
```

对 cpu_relex 的调用使用了一个特定于体系的方式来说，你此时没有在用处理器做事情. 在许多系统中它根本不做任何事; 在对称多线程(" 超线程") 系统中，可能让出核心给其他线程. 在如何情况下，无论何时有可能，这个方法应当明确地避免. 我们展示它是因为偶尔你可能想运行这个代码来更好理解其他代码的内幕.

我们来看一下这个代码如何工作. 这个循环被保证能工作因为 jiffies 被内核头文件声明做易失性的，并且因此，在任何时候 C 代码寻址它时都从内存中获取. 尽管技术上正确(它如同设计的一样工作)，这种忙等待严重地降低了系统性能. 如果你不配置你的内核为抢占操作，这个循环在延时期间完全锁住了处理器; 调度器永远不会抢占一个在内核中运行的进程，并且计算机看起来完全死掉直到时间 j1 到时. 这个问题如果你运行一个可抢占的内核时会改善一点，因为，除非这个代码正持有一个锁，处理器的一些时间可以被其他用途获得. 但是，忙等待在可抢占系统中仍然是昂贵的.

更坏的是，当你进入循环时如果中断碰巧被禁止，jiffies 将不会被更新，并且 while 条件永远保持真. 运行一个抢占的内核也不会有帮助，并且你将被迫去击打大红按钮.

这个延时代码的实现可拿到, 如同下列的, 在 jit 模块中. 模块创建的这些 /proc/jit* 文件每次你读取一行文本就延时一整秒, 并且这些行保证是每个 20 字节. 如果你想测试忙等待代码, 你可以读取 /proc/jitbusy, 每当它返回一行它忙-循环一秒.

为确保读, 最多, 一行(或者几行) 一次从 /proc/jitbusy. 简化的注册 /proc 文件的内核机制反复调用 read 方法来填充用户请求的数据缓存. 因此, 一个命令, 例如 cat /proc/jitbusy, 如果它一次读取 4KB, 会冻住计算机 205 秒.

推荐的读 /proc/jitbusy 的命令是 dd bs=200 < /proc/jitbusy, 可选地同时指定块数目. 文件返回的每 20-字节 的行表示 jiffy 计数器已有的值, 在延时之前和延时之后. 这是一个例子运行在一个其他方面无负担的计算机上:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1686518 1687518
 1687519 1688519
 1688520 1689520
 1689520 1690520
 1690521 1691521
```

看来都挺好: 延时精确地是 1 秒 (1000 jiffies), 并且下一个 read 系统调用在上一个结束后立刻开始. 但是让我们看看在一个有大量 CPU-密集型进程在运行(并且是非抢占内核)的系统上会发生什么:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1911226 1912226
 1913323 1914323
 1919529 1920529
 1925632 1926632
 1931835 1932835
```

这里, 每个 read 系统调用精确地延时 1 秒, 但是内核耗费多过 5 秒在调度 dd 进程以便它可以发出下一个系统调用之前. 在一个多任务系统就期望是这样; CPU 时间在所有运行的进程间共享, 并且一个 CPU-密集型 进程有它的动态减少的优先级. (调度策略的讨论在本书范围之外).

上面所示的在负载下的测试已经在运行 load50 例子程序中进行了. 这个程序派生出许多什么都不做的进程, 但是以一种 CPU-密集的方式来做. 这个程序是伴随本书的例子文件的一部分, 并且缺省是派生 50 个进程, 尽管这个数字可以在命令行指定. 在本章, 以及在本书其他部分, 使用一个有负载的系统的测试已经用 load50 在一个其他方面空闲的计算机上运行来进行了.

如果你在运行一个可抢占内核时重复这个命令, 你会发现没有显著差别在一个其他方面空闲的 CPU 上以及下面的在负载下的行为:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 14940680 14942777
 14942778 14945430
```

```
 14945431 14948491
 14948492 14951960
 14951961 14955840
```

这里, 没有显著的延时在一个系统调用的末尾和下一个的开始之间, 但是单独的延时远远比 1 秒长: 直到 3.8 秒在展示的例子中并且随时间上升. 这些值显示了进程在它的延时当中被中断, 调度其他的进程. 系统调用之间的间隙不是唯一的这个进程的调度选项, 因此没有特别的延时在那里可以看到.

7.3.1.2. 让出处理器

如我们已见到的, 忙等待强加了一个重负载给系统总体; 我们乐意找出一个更好的技术. 想到的第一个改变是明确地释放 CPU 当我们对其不感兴趣时. 这是通过调用调度函数而实现地, 在 <linux/sched.h> 中声明:

```
while (time_before(jiffies, j1)) {
    schedule();
}
```

这个循环可以通过读取 /proc/jitsched 如同我们上面读 /proc/jitbusy 一样来测试. 但是, 还是不够优化. 当前进程除了释放 CPU 不作任何事情, 但是它保留在运行队列中. 如果它是唯一的可运行进程, 实际上它运行(它调用调度器来选择同一个进程, 进程又调用调度器, 这样下去). 换句话说, 机器的负载(在运行的进程的平均数) 最少是 1, 并且空闲任务 (进程号 0, 也称为对换进程, 由于历史原因) 从不运行. 尽管这个问题可能看来无关, 在计算机是空闲时运行空闲任务减轻了处理器工作负载, 降低它的温度以及提高它的生命期, 同时电池的使用时间如果这个计算机是你的膝上机. 更多的, 因为进程实际上在延时中执行, 它所耗费的时间都可以统计.

/proc/jitsched 的行为实际上类似于运行 /proc/jitbusy 在一个抢占的内核下. 这是一个例子运行, 在一个无负载的系统:

```
phon% dd bs=20 count=5 < /proc/jitsched
 1760205 1761207
 1761209 1762211
 1762212 1763212
 1763213 1764213
 1764214 1765217
```

有趣的是要注意每次 read 有时结束于等待比要求的多几个时钟嘀哒. 这个问题随着系统变忙会变得越来越坏, 并且驱动可能结束于等待长于期望的时间. 一旦一个进程使用调度来释放处理器, 无法保证进程将拿回处理器在任何时间之后. 因此, 以这种方式调用调度器对于驱动的需求不是一个安全的解决方法, 另外对计算机系统整体是不好的. 如果你在运行 load50 时测试 jitsched, 你可以见到关联到每一行的延时被扩充了几秒, 因为当定时超时的时候其他进程在使用 CPU .

7.3.1.3. 超时

到目前为止所展示的次优化的延时循环通过查看 jiffy 计数器而不告诉任何人来工作. 但是最好的实现一个延时的方法, 如你可能猜想的, 常常是请求内核为你做. 有 2 种方法来建立一个基于 jiffy 的超时, 依赖于是否你的驱动在等待其他的事件.

如果你的驱动使用一个等待队列来等待某些其他事件, 但是你也想确保它在一个确定时间段内运行, 可以使用 wait_event_timeout 或者 wait_event_interruptible_timeout:

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long
timeout);
```

这些函数在给定队列上睡眠, 但是它们在超时(以 jiffies 表示)到后返回. 因此, 它们实现一个限定的睡眠不会一直睡下去. 注意超时值表示要等待的 jiffies 数, 不是一个绝对时间值. 这个值由一个有符号的数表示, 因为它有时是一个相减运算的结果, 尽管这些函数如果提供的超时值是负值通过一个 printk 语句抱怨. 如果超时到, 这些函数返回 0; 如果这个进程被其他事件唤醒, 它返回以 jiffies 表示的剩余超时值. 返回值从不会是负值, 甚至如果延时由于系统负载而比期望的值大.

/proc/jitqueue 文件展示了一个基于 wait_event_interruptible_timeout 的延时, 结果这个模块没有事件来等待, 并且使用 0 作为一个条件:

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

当读取 /proc/jitqueue 时, 观察到的行为近乎优化的, 即便在负载下:

```
phon% dd bs=20 count=5 < /proc/jitqueue
 2027024  2028024
 2028025  2029025
 2029026  2030026
 2030027  2031027
 2031028  2032028
```

因为读进程当等待超时(上面是 dd)不在运行队列中, 你看不到表现方面的差别, 无论代码是否运行在一个抢占内核中.

wait_event_timeout 和 wait_event_interruptible_timeout 被设计为有硬件驱动存在, 这里可以用任何一种方法来恢复执行: 或者有人调用 wake_up 在等待队列上, 或者超时到. 这不适用于 jitqueue, 因为没人在等待队列上调用 wake_up (毕竟, 没有其他代码知道它), 因此这个进程当超时到时一直唤醒. 为适应这个特别的情况, 这里你想延后执行不等待特定事件, 内核提供了 schedule_timeout 函数, 因此你可以避免声明和使用一个多余的等待队列头:

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

这里，timeout 是要延时的 jiffies 数. 返回值是 0 除非这个函数在给定的 timeout 流失前返回(响应一个信号). schedule_timeout 请求调用者首先设置当前的进程状态，因此一个典型调用看来如此：

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

前面的行(来自 /proc/jitschedto) 导致进程睡眠直到经过给定的时间. 因为 wait_event_interruptible_timeout 在内部依赖 schedule_timeout，我们不会费劲显示 jitschedto 返回的数，因为它们和 jitqueue 的相同. 再一次，不值得有一个额外的时间间隔在超时到和你的进程实际被调度来执行之间.

在刚刚展示的例子中，第一行调用 set_current_state 来设定一些东西以便调度器不会再次运行当前进程，直到超时将它置回 TASK_RUNNING 状态. 为获得一个不可中断的延时，使用 TASK_UNINTERRUPTIBLE 代替. 如果你忘记改变当前进程的状态，调用 schedule_time 如同调用 shcedule(即，jitsched 的行为)，建立一个不用的定时器.

如果你想使用这 4 个 jit 文件在不同的系统情况下或者不同的内核，或者尝试其他的方式来延后执行，你可能想配置延时量当加载模块时通过设定延时模块参数.

7.3.2. 短延时

当一个设备驱动需要处理它的硬件的反应时间，涉及到的延时常常是最多几个毫秒. 在这个情况下，依靠时钟嘀哒显然不对路.

The kernel functions ndelay, udelay, and mdelay serve well for short delays, delaying execution for the specified number of nanoseconds, microseconds, or milliseconds respectively.* Their prototypes are: * The u in udelay represents the Greek letter mu and stands for micro.

内核函数 ndelay，udelay，以及 mdelay 对于短延时好用，分别延后执行指定的纳秒数，微秒数或者毫秒数. [27][27]它们的原型是：

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

这些函数的实际实现在 <asm/delay.h>，是体系特定的，并且有时建立在一个外部函数上. 每个体系都实现 udelay，但是其他的函数可能或者不可能定义；如果它们没有定义，<linux/delay.h> 提供一个缺省的基于 udelay 的版本. 在所有的情况中，获得的延时至

[27] [27] udelay 中的 u 表示希腊字母 mu 并且代表 micro.

少是要求的值，但可能更多；实际上，当前没有平台获得了纳秒的精度，尽管有几个提供了次微秒的精度．延时多于要求的值通常不是问题，因为驱动中的短延时常常需要等待硬件，并且这个要求是等待至少一个给定的时间流失．

udelay 的实现（ 可能 ndelay 也是）使用一个软件循环基于在启动时计算的处理器速度，使用整数变量 loos_per_jiffy．如果你想看看实际的代码，但是，小心 x86 实现是相当复杂的一个因为它使用的不同的时间源，基于什么 CPU 类型在运行代码．

为避免在循环计算中整数溢出，udelay 和 ndelay 强加一个上限给传递给它们的值．如果你的模块无法加载和显示一个未解决的符号，__bad_udelay，这意味着你使用太大的参数调用 udleay．注意，但是，编译时检查只对常量进行并且不是所有的平台实现它．作为一个通用的规则，如果你试图延时几千纳秒，你应当使用 udelay 而不是 ndelay；类似地，毫秒规模的延时应当使用 mdelay 完成而不是一个更细粒度的函数．

重要的是记住这 3 个延时函数是忙等待；其他任务在时间流失时不能运行．因此，它们重复，尽管在一个不同的规模上，jitbusy 的做法．因此，这些函数应当只用在没有实用的替代时．

有另一个方法获得毫秒(和更长)延时而不用涉及到忙等待．文件 <linux/delay.h> 声明这些函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

前 2 个函数使调用进程进入睡眠给定的毫秒数．一个对 msleep 的调用是不可中断的；你能确保进程睡眠至少给定的毫秒数．如果你的驱动位于一个等待队列并且你想唤醒来打断睡眠，使用 msleep_interruptible．从 msleep_interruptible 的返回值正常地是 0；如果，但是，这个进程被提早唤醒，返回值是在初始请求睡眠周期中剩余的毫秒数．对 ssleep 的调用使进程进入一个不可中断的睡眠给定的秒数．

通常，如果你能够容忍比请求的更长的延时，你应当使用 schedule_timeout，msleep，或者 ssleep．

7.4. 内核定时器

无论何时你需要调度一个动作以后发生，而不阻塞当前进程直到到时，内核定时器是给你的工具．这些定时器用来调度一个函数在将来一个特定的时间执行，基于时钟嘀哒，并且可用作各类任务；例如，当硬件无法发出中断时，查询一个设备通过在定期的间隔内检查它的状态．其他的内核定时器的典型应用是关闭软驱马达或者结束另一个长期终止的操作．在这种情况下，延后来自 close 的返回将强加一个不必要(并且吓人的)开销在应用程序上．最后，内核自身使用定时器在几个情况下，包括实现 schedule_timeout．

一个内核定时器是一个数据结构，它指导内核执行一个用户定义的函数使用一个用户定义的参数在一个用户定义的时间．这个实现位于 <linux/timer.h> 和 kernel/timer.c 并且在"内核定时器"一节中详细介绍．

被调度运行的函数几乎确定不会在注册它们的进程在运行时运行. 它们是, 相反, 异步运行. 直到现在, 我们在我们的例子驱动中已经做的任何事情已经在执行系统调用的进程上下文中运行. 当一个定时器运行时, 但是, 这个调度进程可能睡眠, 可能在不同的一个处理器上运行, 或者很可能已经一起退出.

这个异步执行类似当发生一个硬件中断时所发生的(这在第 10 章详细讨论). 实际上, 内核定时器被作为一个"软件中断"的结果而实现. 当在这种原子上下文运行时, 你的代码易受到多个限制. 定时器函数必须是原子的以所有的我们在第 1 章"自旋锁和原子上下文"一节中曾讨论过的方式, 但是有几个附加的问题由于缺少一个进程上下文而引起的. 我们将介绍这些限制; 在后续章节的几个地方将再次看到它们. 循环被调用因为原子上下文的规则必须认真遵守, 否则系统会发现自己陷入大麻烦中.

为能够被执行, 多个动作需要进程上下文. 当你在进程上下文之外(即, 在中断上下文), 你必须遵守下列规则:

- 没有允许存取用户空间. 因为没有进程上下文, 没有和任何特定进程相关联的到用户空间的途径.
- 这个 current 指针在原子态没有意义, 并且不能使用因为相关的代码没有和已被中断的进程的联系.
- 不能进行睡眠或者调度. 原子代码不能调用 schedule 或者某种 wait_event, 也不能调用任何其他可能睡眠的函数. 例如, 调用 kmalloc(..., GFP_KERNEL) 是违犯规则的. 旗标也必须不能使用因为它们可能睡眠.

内核代码能够告知是否它在中断上下文中运行, 通过调用函数 in_interrupt(), 它不要参数并且如果处理器当前在中断上下文运行就返回非零, 要么硬件中断要么软件中断.

一个和 in_interrupt() 相关的函数是 in_atomic(). 它的返回值是非零无论何时调度被禁止; 这包含硬件和软件中断上下文以及任何持有自旋锁的时候. 在后一种情况, current 可能是有效的, 但是存取用户空间被禁止, 因为它能导致调度发生. 无论何时你使用 in_interrupt(), 你应当真正考虑是否 in_atomic 是你实际想要的. 2 个函数都在 <asm/hardirq.h> 中声明.

内核定时器的另一个重要特性是一个任务可以注册它本身在后面时间重新运行. 这是可能的, 因为每个 timer_list 结构在运行前从激活的定时器链表中去连接, 并且因此能够马上在其他地方被重新连接. 尽管反复重新调度相同的任务可能表现为一个无意义的操作, 有时它是有用的. 例如, 它可用作实现对设备的查询.

也值得了解在一个 SMP 系统, 定时器函数被注册时相同的 CPU 来执行, 为在任何可能的时候获得更好的缓存局部特性. 因此, 一个重新注册它自己的定时器一直运行在同一个 CPU.

不应当被忘记的定时器的一个重要特性是, 它们是一个潜在的竞争条件的源, 即便在一个单处理器系统. 这是它们与其他代码异步运行的一个直接结果. 因此, 任何被定时器函数存取的数据结构应当保护避免并发存取, 要么通过原子类型(在第 1 章的"原子变量"一节) 要么使用自旋锁(在第 9 章讨论).

7.4.1. 定时器 API

内核提供给驱动许多函数来声明，注册，以及去除内核定时器. 下列的引用展示了基本的代码块:

```
#include <linux/timer.h>
struct timer_list
{
        /* ... */
        unsigned long expires;
        void (*function)(unsigned long);
        unsigned long data;
};
void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);

void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

这个数据结构包含比曾展示过的更多的字段，但是这 3 个是打算从定时器代码自身以外被存取的. 这个 expires 字段表示定时器期望运行的 jiffies 值; 在那个时间，这个 function 函数被调用使用 data 作为一个参数. 如果你需要在参数中传递多项，你可以捆绑它们作为一个单个数据结构并且传递一个转换为 unsiged long 的指针，在所有支持的体系上的一个安全做法并且在内存管理中相当普遍(如同 15 章中讨论的). expires 值不是一个 jiffies_64 项因为定时器不被期望在将来很久到时，并且 64-位操作在 32-位平台上慢.

这个结构必须在使用前初始化. 这个步骤保证所有的成员被正确建立，包括那些对调用者不透明的. 初始化可以通过调用 init_timer 或者 安排 TIMER_INITIALIZER 给一个静态结构，根据你的需要. 在初始化后，你可以改变 3 个公共成员在调用 add_timer 前. 为在到时前禁止一个已注册的定时器，调用 del_timer.

jit 模块包括一个例子文件，/proc/jitimer (为 "just in timer")，它返回一个头文件行以及 6 个数据行. 这些数据行表示当前代码运行的环境; 第一个由读文件操作产生并且其他的由定时器. 下列的输出在编译内核时被记录:

```
phon% cat /proc/jitimer
 time delta  inirq  pid  cpu command
 33565837  0  0  1269  0  cat
 33565847  10  1  1271  0  sh
 33565857  10  1  1273  0  cpp0
 33565867  10  1  1273  0  cpp0
 33565877  10  1  1274  0  cc1
 33565887  10  1  1274  0  cc1
```

在这个输出，time 字段是代码运行时的 jiffies 值，delta 是自前一行的 jiffies 改变值，inirq 是由 in_interrupt 返回的布尔值，pid 和 command 指的是当前进程，以及 cpu 是在使用的 CPU 的数目(在单处理器系统上一直为 0).

如果你读 /proc/jitimer 当系统无负载时，你会发现定时器的上下文是进程 0，空闲任务，它被称为"对换进程"只要由于历史原因.

用来产生 /proc/jitimer 数据的定时器是缺省每 10 jiffies 运行一次，但是你可以在加载模块时改变这个值通过设置 tdelay (timer delay) 参数.

下面的代码引用展示了 jit 关于 jitimer 定时器的部分. 当一个进程试图读取我们的文件，我们建立这个定时器如下:

```
unsigned long j = jiffies;
/* fill the data for our timer function */
data->prevjiffies = j;

data->buf = buf2;
data->loops = JIT_ASYNC_LOOPS;

/* register the timer */
data->timer.data = (unsigned long)data;
data->timer.function = jit_timer_fn;
data->timer.expires = j + tdelay; /* parameter */
add_timer(&data->timer);

/* wait for the buffer to fill */
wait_event_interruptible(data->wait, !data->loops);
```

The actual timer function looks like this:

```
void jit_timer_fn(unsigned long arg)
{
        struct jit_data *data = (struct jit_data *)arg;
        unsigned long j = jiffies;
        data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n",
                             j, j - data->prevjiffies, in_interrupt() ? 1 : 0,
                             current->pid, smp_processor_id(), current->comm);
        if (--data->loops) {
                data->timer.expires += tdelay;
                data->prevjiffies = j;
                add_timer(&data->timer);

        } else {
                wake_up_interruptible(&data->wait);
        }
}
```

定时器 API 包括几个比上面介绍的那些更多的功能. 下面的集合是完整的核提供的函数列表:

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

更新一个定时器的超时时间, 使用一个超时定时器的一个普通的任务(再一次, 关马达软驱定时器是一个典型例子). mod_timer 也可被调用于非激活定时器, 那里你正常地使用 add_timer.

int del_timer_sync(struct timer_list *timer);

如同 del_timer 一样工作, 但是还保证当它返回时, 定时器函数不在任何 CPU 上运行. del_timer_sync 用来避免竞争情况在 SMP 系统上, 并且在 UP 内核中和 del_timer 相同. 这个函数应当在大部分情况下比 del_timer 更首先使用. 这个函数可能睡眠如果它被从非原子上下文调用, 但是在其他情况下会忙等待. 要十分小心调用 del_timer_sync 当持有锁时; 如果这个定时器函数试图获得同一个锁, 系统会死锁. 如果定时器函数重新注册自己, 调用者必须首先确保这个重新注册不会发生; 这常常同设置一个" 关闭 "标志来实现, 这个标志被定时器函数检查.

int timer_pending(const struct timer_list * timer);

返回真或假来指示是否定时器当前被调度来运行, 通过调用结构的其中一个不透明的成员.

7.4.2. 内核定时器的实现

为了使用它们, 尽管你不会需要知道内核定时器如何实现, 这个实现是有趣的, 并且值得看一下它们的内部.

定时器的实现被设计来符合下列要求和假设:

- 定时器管理必须尽可能简化.
- 设计应当随着激活的定时器数目上升而很好地适应.
- 大部分定时器在几秒或最多几分钟内到时, 而带有长延时的定时器是相当少见.
- 一个定时器应当在注册它的同一个 CPU 上运行.

由内核开发者想出的解决方法是基于一个每-CPU 数据结构. 这个 timer_list 结构包括一个指针指向这个的数据结构在它的 base 成员. 如果 base 是 NULL, 这个定时器没有被调用运行; 否则, 这个指针告知哪个数据结构(并且, 因此, 哪个 CPU)运行它. 每-CPU 数据项在第 8 章的"每-CPU 变量"一节中描述.

无论何时内核代码注册一个定时器(通过 add_timer 或者 mod_timer), 操作最终由 internal_add_timer 进行(在 kernel/timer.c), 它依次添加新定时器到一个双向定时器链表在一个关联到当前 CPU 的"层叠表" 中.

这个层叠表象这样工作: 如果定时器在下一个 0 到 255 jiffies 内到时, 它被添加到专供短时定时器 256 列表中的一个上, 使用 expires 成员的最低有效位. 如果它在将来更久时间到时(但是在 16,384 jiffies 之前), 它被添加到基于 expires 成员的 9 - 14 位的 64 个列表中一个. 对于更长的定时器, 同样的技巧用在 15 - 20 位, 21 - 26 位, 和 27 - 31 位. 带有一个指向将来还长时间的 expires 成员的定时器(一些只可能发生在 64-位 平台上的事情) 被使用一个延时值 0xffffffff 进行哈希处理, 并且带有在过

去到时的定时器被调度来在下一个时钟嘀哒运行. (一个已经到时的定时器模拟有时在高负载情况下被注册, 特别的是如果你运行一个可抢占内核).

当触发 __run_timers, 它为当前定时器嘀哒执行所有挂起的定时器. 如果 jiffies 当前是 256 的倍数, 这个函数还重新哈希处理一个下一级别的定时器列表到 256 短期列表, 可能地层叠一个或多个别的级别, 根据 jiffies 的位表示.

这个方法, 虽然第一眼看去相当复杂, 在几个和大量定时器的时候都工作得很好. 用来管理每个激活定时器的时间独立于已经注册的定时器数目并且限制在几个对于它的 expires 成员的二进制表示的逻辑操作上. 关联到这个实现的唯一的开销是给 512 链表头的内存(256 短期链表和 4 组 64 更长时间的列表) -- 即 4 KB 的容量.

函数 __run_timers, 如同 /proc/jitimer 所示, 在原子上下文运行. 除了我们已经描述过的限制, 这个带来一个有趣的特性: 定时器刚好在合适的时间到时, 甚至你没有运行一个可抢占内核, 并且 CPU 在内核空间忙. 你可以见到发生了什么当你在后台读 /proc/jitbusy 时以及在前台 /proc/jitimer. 尽管系统看来牢固地被锁住被这个忙等待系统调用, 内核定时器照样工作地不错.

但是, 记住, 一个内核定时器还远未完善, 因为它受累于 jitter 和 其他由硬件中断引起怪物, 还有其他定时器和其他异步任务. 虽然一个关联到简单数字 I/O 的定时器对于一个如同运行一个步进马达或者其他业余电子设备等简单任务是足够的, 它常常是不合适在工业环境中的生产系统. 对于这样的任务, 你将最可能需要依赖一个实时内核扩展.

7.5. Tasklets 机制

另一个有关于定时问题的内核设施是 tasklet 机制. 它大部分用在中断管理(我们将在第 10 章再次见到).

tasklet 类似内核定时器在某些方面. 它们一直在中断时间运行, 它们一直运行在调度它们的同一个 CPU 上, 并且它们接收一个 unsigned long 参数. 不象内核定时器, 但是, 你无法请求在一个指定的时间执行函数. 通过调度一个 tasklet, 你简单地请求它在以后的一个由内核选择的时间执行. 这个行为对于中断处理特别有用, 那里硬件中断必须被尽快处理, 但是大部分的时间管理可以安全地延后到以后的时间. 实际上, 一个 tasket, 就象一个内核定时器, 在一个"软中断"的上下文中执行(以原子模式), 在使能硬件中断时执行异步任务的一个内核机制.

一个 tasklet 存在为一个时间结构, 它必须在使用前被初始化. 初始化能够通过调用一个特定函数或者通过使用某些宏定义声明结构:

```
#include <linux/interrupt.h>
struct tasklet_struct {
 /* ... */

void (*func)(unsigned long);
 unsigned long data;
};
```

```
void tasklet_init(struct tasklet_struct *t,
 void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

tasklet 提供了许多有趣的特色:

- 一个 tasklet 能够被禁止并且之后被重新使能; 它不会执行直到它被使能与被禁止相同的的次数.
- 如同定时器, 一个 tasklet 可以注册它自己.
- 一个 tasklet 能被调度来执行以正常的优先级或者高优先级. 后一组一直是首先执行.
- taslet 可能立刻运行, 如果系统不在重载下, 但是从不会晚于下一个时钟嘀哒.
- 一个 tasklet 可能和其他 tasklet 并发, 但是对它自己是严格地串行的 -- 同样的 tasklet 从不同时运行在超过一个处理器上. 同样, 如已经提到的, 一个 tasklet 常常在调度它的同一个 CPU 上运行.

jit 模块包括 2 个文件, /proc/jitasklet 和 /proc/jitasklethi, 它返回和在"内核定时器"一节中介绍过的 /proc/jitimer 同样的数据. 当你读其中一个文件时, 你取回一个 header 行. 第一个数据行描述了调用进程的上下文, 并且其他的行描述了一个 tasklet 过程连续运行的上下文. 这是一个在编译一个内核时的运行例子:

```
phon% cat /proc/jitasklet
 time delta inirq pid cpu command
 6076139 0 0 4370 0 cat
 6076140 1 1 4368 0 cc1
 6076141 1 1 4368 0 cc1
 6076141 0 1 2 0 ksoftirqd/0
 6076141 0 1 2 0 ksoftirqd/0
 6076141 0 1 2 0 ksoftirqd/0
```

如同由上面数据所确定的, tasklet 在下一个时间嘀哒内运行只要 CPU 在忙于运行一个进程, 但是它立刻被运行当 CPU 处于空闲. 内核提供了一套 ksoftirqd 内核线程, 每个 CPU 一个, 只是来运行 "软件中断" 处理, 就像 tasklet_action 函数. 因此, tasklet 的最后 3 个运行在关联到 CPU 0 的 ksoftirqd 内核线程的上下文中发生. jitasklethi 的实现使用了一个高优先级 tasklet, 在马上要来的函数列表中解释.

jit 中实现 /proc/jitasklet 和 /proc/jittasklethi 的实际代码与 /proc/jitimer 的实现代码几乎是一致的, 但是它使用 tasklet 调用代替那些定时器. 下面的列表详细展开了 tasklet 结构已被初始化后的内核对 tasklet 的接口:

```
void tasklet_disable(struct tasklet_struct *t);
```

这个函数禁止给定的 tasklet. tasklet 可能仍然被 tasklet_schedule 调度, 但是它的执行被延后直到这个 tasklet 被再次使能. 如果这个 tasklet 当前在运行, 这个函数忙等待直到这个 tasklet 退出; 因此, 在调用 tasklet_disable 后, 你可以确保这个 tasklet 在系统任何地方都不在运行.

void tasklet_disable_nosync(struct tasklet_struct *t);

禁止这个 tasklet，但是没有等待任何当前运行的函数退出. 当它返回，这个 tasklt 被禁止并且不会在以后被调度直到重新使能，但是它可能仍然运行在另一个 CPU 当这个函数返回时.

void tasklet_enable(struct tasklet_struct *t);

使能一个之前被禁止的 tasklet. 如果这个 tasklet 已经被调度，它会很快运行. 一个对 tasklet_enable 的调用必须匹配每个对 tasklet_disable 的调用，因为内核跟踪每个 tasklet 的"禁止次数".

void tasklet_schedule(struct tasklet_struct *t);

调度 tasklet 执行. 如果一个 tasklet 被再次调度在它有机会运行前，它只运行一次. 但是，如果他在运行中被调度，它在完成后再次运行; 这保证了在其他事件被处理当中发生的事件收到应有的注意. 这个做法也允许一个 tasklet 重新调度它自己.

void tasklet_hi_schedule(struct tasklet_struct *t);

调度 tasklet 在更高优先级执行. 当软中断处理运行时，它处理高优先级 tasklet 在其他软中断之前，包括"正常的" tasklet. 理想地，只有具有低响应周期要求(例如填充音频缓冲)应当使用这个函数，为避免其他软件中断处理引入的附加周期. 实际上，/proc/jitasklethi 没有显示可见的与 /proc/jitasklet 的区别.

void tasklet_kill(struct tasklet_struct *t);

这个函数确保了这个 tasklet 没被再次调度来运行; 它常常被调用当一个设备正被关闭或者模块卸载时. 如果这个 tasklet 被调度来运行，这个函数等待直到它已执行. 如果这个 tasklet 重新调度它自己，你必须阻止在调用 tasklet_kill 前它重新调度它自己，如同使用 del_timer_sync.

tasklet 在 kernel/softirq.c 中实现. 2 个 tasklet 链表(正常和高优先级)被声明为每-CPU 数据结构，使用和内核定时器相同的 CPU-亲和 机制. 在 tasklet 管理中的数据结构是简单的链表，因为 tasklet 没有内核定时器的分类请求.

7.6. 工作队列

工作队列是，表面上看，类似于 taskets; 它们允许内核代码来请求在将来某个时间调用一个函数. 但是，有几个显著的不同在这 2 个之间，包括:

- tasklet 在软件中断上下文中运行的结果是所有的 tasklet 代码必须是原子的. 相反，工作队列函数在一个特殊内核进程上下文运行; 结果，它们有更多的灵活性. 特别地，工作队列函数能够睡眠.

- tasklet 常常在它们最初被提交的处理器上运行. 工作队列以相同地方式工作, 缺省地.
- 内核代码可以请求工作队列函数被延后一个明确的时间间隔.

两者之间关键的不同是 tasklet 执行的很快, 短时期, 并且在原子态, 而工作队列函数可能有高周期但是不需要是原子的. 每个机制有它适合的情形.

工作队列有一个 struct workqueue_struct 类型, 在 <linux/workqueue.h> 中定义. 一个工作队列必须明确的在使用前创建, 使用一个下列的 2 个函数:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

每个工作队列有一个或多个专用的进程("内核线程"), 它运行提交给这个队列的函数. 如果你使用 create_workqueue, 你得到一个工作队列它有一个专用的线程在系统的每个处理器上. 在很多情况下, 所有这些线程是简单的过度行为; 如果一个单个工作者线程就足够, 使用 create_singlethread_workqueue 来代替创建工作队列

提交一个任务给一个工作队列, 你需要填充一个 work_struct 结构. 这可以在编译时完成, 如下:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

这里 name 是声明的结构名称, function 是从工作队列被调用的函数, 以及 data 是一个传递给这个函数的值. 如果你需要建立 work_struct 结构在运行时, 使用下面 2 个宏定义:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK 做更加全面的初始化结构的工作; 你应当在第一次建立结构时使用它. PREPARE_WORK 做几乎同样的工作, 但是它不初始化用来连接 work_struct 结构到工作队列的指针. 如果有任何的可能性这个结构当前被提交给一个工作队列, 并且你需要改变这个队列, 使用 PREPARE_WORK 而不是 INIT_WORK.

有 2 个函数来提交工作给一个工作队列:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct
*work, unsigned long delay);
```

每个都添加工作到给定的队列. 如果使用 queue_delay_work, 但是, 实际的工作没有进行直到至少 delay jiffies 已过去. 从这些函数的返回值是 0 如果工作被成功加入到队列; 一个非零结果意味着这个 work_struct 结构已经在队列中等待, 并且第 2 次没有加入.

在将来的某个时间, 这个工作函数将被使用给定的 data 值来调用. 这个函数将在工作者线程的上下文运行, 因此它可以睡眠如果需要 -- 尽管你应当知道这个睡眠可能怎样影响

提交给同一个工作队列的其他任务. 这个函数不能做的是, 但是, 是存取用户空间. 因为它在一个内核线程中运行, 完全没有用户空间来存取.

如果你需要取消一个挂起的工作队列入口, 你可以调用:

```
int cancel_delayed_work(struct work_struct *work);
```

返回值是非零如果这个入口在它开始执行前被取消. 内核保证给定入口的执行不会在调用 cancel_delay_work 后被初始化. 如果 cancel_delay_work 返回 0, 但是, 这个入口可能已经运行在一个不同的处理器, 并且可能仍然在调用 cancel_delayed_work 后在运行. 要绝对确保工作函数没有在 cancel_delayed_work 返回 0 后在任何地方运行, 你必须跟随这个调用来调用:

```
void flush_workqueue(struct workqueue_struct *queue);
```

在 flush_workqueue 返回后, 没有在这个调用前提交的函数在系统中任何地方运行.

当你用完一个工作队列, 你可以去掉它, 使用:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

7.6.1. 共享队列

一个设备驱动, 在许多情况下, 不需要它自己的工作队列. 如果你只偶尔提交任务给队列, 简单地使用内核提供的共享的, 缺省的队列可能更有效. 如果你使用这个队列, 但是, 你必须明白你将和别的在共享它. 从另一个方面说, 这意味着你不应当长时间独占队列(无长睡眠), 并且可能要更长时间它们轮到处理器.

jiq ("just in queue") 模块输出 2 个文件来演示共享队列的使用. 它们使用一个单个 work_struct structure, 这个结构这样建立:

```
static struct work_struct jiq_work;
    /* this line is in jiq_init() */
 INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

当一个进程读 /proc/jiqwq, 这个模块不带延迟地初始化一系列通过共享的工作队列的路线.

```
int schedule_work(struct work_struct *work);
```

注意, 当使用共享队列时使用了一个不同的函数; 它只要求 work_struct 结构作为一个参数. 在 jiq 中的实际代码看来如此:

```
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
schedule_work(&jiq_work);
schedule();
finish_wait(&jiq_wait, &wait);
```

这个实际的工作函数打印出一行就象 jit 模块所作的，接着，如果需要，重新提交这个 work_structcture 到工作队列中. 在这是 jiq_print_wq 全部:

```
static void jiq_print_wq(void *ptr)
{
        struct clientdata *data = (struct clientdata *) ptr;

        if (! jiq_print (ptr))
                return;

        if (data->delay)
                schedule_delayed_work(&jiq_work, data->delay);
        else
                schedule_work(&jiq_work);
}
```

如果用户在读被延后的设备 (/proc/jiqwqdelay)，这个工作函数重新提交它自己在延后的模式，使用 schedule_delayed_work:

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

如果你看从这 2 个设备的输出，它看来如:

```
% cat /proc/jiqwq
 time   delta preempt  pid cpu command
 1113043  0   0   7   1 events/1
 1113043  0   0   7   1 events/1
 1113043  0   0   7   1 events/1
 1113043  0   0   7   1 events/1
 1113043  0   0   7   1 events/1
% cat /proc/jiqwqdelay
 time   delta preempt  pid cpu command
 1122066  1   0   6   0 events/0

1122067  1   0   6   0 events/0
 1122068  1   0   6   0 events/0
 1122069  1   0   6   0 events/0
 1122070  1   0   6   0 events/0
```

当 /proc/jiqwq 被读，在每行的打印之间没有明显的延迟. 相反，当 /proc/jiqwqdealy 被读时，在每行之间有恰好一个 jiffy 的延时. 在每一种情况，我们看到同样的进程名子被打印；它是实现共享队列的内核线程的名子. CPU 号被打印在斜线后面；我们从不知道当读 /proc 文件时哪个 CPU 会在运行，但是这个工作函数之后将一直运行在同一个处理器.

如果你需要取消一个已提交给工作队列的工作入口，你可以使用 cancel_delayed_work，如上面所述. 刷新共享队列需要一个不同的函数，但是:

```
void flush_scheduled_work(void);
```

因为你不知道别人谁可能使用这个队列，你从不真正知道 flush_schduled_work 返回可能需要多长时间.

7.7. 快速参考

本章介绍了下面的符号.

7.7.1. 时间管理

```
#include <linux/param.h>
HZ
```

HZ 符号指定了每秒产生的时钟嘀哒的数目.

```
#include <linux/jiffies.h>
volatile unsigned long jiffies;
u64 jiffies_64;
```

jiffies_64 变量每个时钟嘀哒时被递增；因此，它是每秒递增 HZ 次. 内核代码几乎常常引用 jiffies，它在 64-位平台和 jiffies_64 相同并且在 32-位平台是它低有效的一半.

```
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

这些布尔表达式以一种安全的方式比较 jiffies，没有万一计数器溢出的问题和不需要存取 jiffies_64.

```
u64 get_jiffies_64(void);
```

获取 jiffies_64 而没有竞争条件.

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

在 jiffies 和其他表示之间转换时间表示.

```
#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
```

```
rdtscll(var32);
```

x86-特定的宏定义来读取时戳计数器. 它们作为 2 半 32-位来读取, 只读低一半, 或者全部读到一个 long long 变量.

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

以平台独立的方式返回时戳计数器. 如果 CPU 没提供时戳特性, 返回 0.

```
#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
```

返回自 Epoch 以来的秒数, 基于 6 个 unsigned int 参数.

```
void do_gettimeofday(struct timeval *tv);
```

返回当前时间, 作为自 Epoch 以来的秒数和微秒数, 用硬件能提供的最好的精度. 在大部分的平台这个解决方法是一个微秒或者更好, 尽管一些平台只提供 jiffies 精度.

```
struct timespec current_kernel_time(void);
```

返回当前时间, 以一个 jiffy 的精度.

7.7.2. 延迟

```
#include <linux/wait.h>
long wait_event_interruptible_timeout(wait_queue_head_t *q, condition, signed long timeout);
```

使当前进程在等待队列进入睡眠, 安装一个以 jiffies 表达的超时值. 使用 schedule_timeout(下面) 给不可中断睡眠.

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

调用调度器, 在确保当前进程在超时到的时候被唤醒后. 调用者首先必须调用 set_curret_state 来使自己进入一个可中断的或者不可中断的睡眠状态.

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

引入一个整数纳秒, 微秒和毫秒的延迟. 获得的延迟至少是请求的值, 但是可能更多. 每个函数的参数必须不超过一个平台特定的限制(常常是几千).

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

使进程进入睡眠给定的毫秒数(或者秒, 如果使 ssleep).

7.7.3. 内核定时器

```
#include <asm/hardirq.h>
int in_interrupt(void);
int in_atomic(void);
```

返回一个布尔值告知是否调用代码在中断上下文或者原子上下文执行. 中断上下文是在一个进程上下文之外, 或者在硬件或者软件中断处理中. 原子上下文是当你不能调度一个中断上下文或者一个持有一个自旋锁的进程的上下文.

```
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
```

这个函数和静态的定时器结构的声明是初始化一个 timer_list 数据结构的 2 个方法.

```
void add_timer(struct timer_list * timer);
```

注册定时器结构来在当前 CPU 上运行.

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

改变一个已经被调度的定时器结构的超时时间. 它也能作为一个 add_timer 的替代.

```
int timer_pending(struct timer_list * timer);
```

宏定义, 返回一个布尔值说明是否这个定时器结构已经被注册运行.

```
void del_timer(struct timer_list * timer);
void del_timer_sync(struct timer_list * timer);
```

从激活的定时器链表中去除一个定时器. 后者保证这定时器当前没有在另一个 CPU 上运行.

7.7.4. Tasklets 机制

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
unsigned long data);
```

前 2 个宏定义声明一个 tasklet 结构, 而 tasklet_init 函数初始化一个已经通过分配或其他方式获得的 tasklet 结构. 第 2 个 DECLARE 宏标识这个 tasklet 为禁止的.

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

禁止和使能一个 tasklet. 每个禁止必须配对一个使能(你可以禁止这个 tasklet 即便它已经被禁止). 函数 tasklet_disable 等待 tasklet 终止如果它在另一个 CPU 上运行. 这个非同步版本不采用这个额外的步骤.

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

调度一个 tasklet 运行, 或者作为一个"正常" tasklet 或者一个高优先级的. 当软中断被执行, 高优先级 tasklets 被首先处理, 而正常 tasklet 最后执行.

```
void tasklet_kill(struct tasklet_struct *t);
```

从激活的链表中去掉 tasklet, 如果它被调度执行. 如同 tasklet_disable, 这个函数可能在 SMP 系统中阻塞等待 tasklet 终止, 如果它当前在另一个 CPU 上运行.

7.7.5. 工作队列

```
#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;
```

这些结构分别表示一个工作队列和一个工作入口.

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);
```

创建和销毁工作队列的函数. 一个对 create_workqueue 的调用创建一个有一个工作者线程在系统中每个处理器上的队列; 相反, create_singlethread_workqueue 创建一个有一个单个工作者进程的工作队列.

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

声明和初始化工作队列入口的宏.

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct
*work, unsigned long delay);
```

从一个工作队列对工作进行排队执行的函数.

```
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
```

使用 cancel_delayed_work 来从一个工作队列中去除入口; flush_workqueue 确保没有工作队列入口在系统中任何地方运行.

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
```

使用共享队列的函数.

第 8 章 分配内存

至此，我们已经使用 kmalloc 和 kfree 来分配和释放内存. Linux 内核提供了更丰富的一套内存分配原语，但是. 在本章，我们查看在设备驱动中使用内存的其他方法和如何优化你的系统的内存资源. 我们不涉及不同的体系实际上如何管理内存. 模块不牵扯在分段，分页等问题中，因为内核提供一个统一的内存管理驱动接口. 另外，我们不会在本章描述内存管理的内部细节，但是推迟在 15 章.

8.1. kmalloc 的真实故事

kmalloc 分配引擎是一个有力的工具并且容易学习因为它对 malloc 的相似性. 这个函数快(除非它阻塞)并且不清零它获得的内存；分配的区仍然持有它原来的内容.[28][28] 分配的区也是在物理内存中连续. 在下面几节，我们详细讨论 kmalloc，因此你能比较它和我们后来要讨论的内存分配技术.

8.1.1. flags 参数

记住 kmalloc 原型是:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

给 kmalloc 的第一个参数是要分配的块的大小. 第 2 个参数，分配标志，非常有趣，因为它以几个方式控制 kmalloc 的行为.

最一般使用的标志，GFP_KERNEL，意思是这个分配((内部最终通过调用 __get_free_pages 来进行，它是 GFP_ 前缀的来源) 代表运行在内核空间的进程而进行的. 换句话说，这意味着调用函数是代表一个进程在执行一个系统调用. 使用 GFP_KENRL 意味着 kmalloc 能够使当前进程在少内存的情况下睡眠来等待一页. 一个使用 GFP_KERNEL 来分配内存的函数必须，因此，是可重入的并且不能在原子上下文中运行. 当当前进程睡眠，内核采取正确的动作来定位一些空闲内存，或者通过刷新缓存到磁盘或者交换出去一个用户进程的内存.

GFP_KERNEL 不一直是使用的正确分配标志；有时 kmalloc 从一个进程的上下文的外部调用. 例如，这类的调用可能发生在中断处理，tasklet，和内核定时器中. 在这个情况下，当前进程不应当被置为睡眠，并且驱动应当使用一个 GFP_ATOMIC 标志来代替. 内核正常地试图保持一些空闲页以便来满足原子的分配. 当使用 GFP_ATOMIC 时，kmalloc 能够使用甚至最后一个空闲页. 如果这最后一个空闲页不存在，但是，分配失败.

其他用来代替或者增添 GFP_KERNEL 和 GFP_ATOMIC 的标志，尽管它们 2 个涵盖大部分设备驱动的需要. 所有的标志定义在 <linux/gfp.h>，并且每个标志用一个双下划线做前

[28] [28] 在其他的之中，这暗含着你应当明确地清零可能暴露给用户空间或者写入设备的内存；否则，你可能冒险将应当保密的信息透露出去.

缀, 例如 __GFP_DMA. 另外, 有符号代表常常使用的标志组合; 这些缺乏前缀并且有时被称为分配优先级. 后者包括:

GFP_ATOMIC

用来从中断处理和进程上下文之外的其他代码中分配内存. 从不睡眠.

GFP_KERNEL

内核内存的正常分配. 可能睡眠.

GFP_USER

用来为用户空间页来分配内存; 它可能睡眠.

GFP_HIGHUSER

如同 GFP_USER, 但是从高端内存分配, 如果有. 高端内存在下一个子节描述.

GFP_NOIO
GFP_NOFS

这个标志功能如同 GFP_KERNEL, 但是它们增加限制到内核能做的来满足请求. 一个 GFP_NOFS 分配不允许进行任何文件系统调用, 而 GFP_NOIO 根本不允许任何 I/O 初始化. 它们主要地用在文件系统和虚拟内存代码, 那里允许一个分配睡眠, 但是递归的文件系统调用会是一个坏注意.

上面列出的这些分配标志可以是下列标志的相或来作为参数, 这些标志改变这些分配如何进行:

__GFP_DMA

这个标志要求分配在能够 DMA 的内存区. 确切的含义是平台依赖的并且在下面章节来解释.

__GFP_HIGHMEM

这个标志指示分配的内存可以位于高端内存.

__GFP_COLD

正常地, 内存分配器尽力返回"缓冲热"的页 -- 可能在处理器缓冲中找到的页. 相反, 这个标志请求一个"冷"页, 它在一段时间没被使用. 它对分配页作 DMA 读是有用的, 此时在处理器缓冲中出现是无用的. 一个完整的对如何分配 DMA 缓存的讨论看"直接内存存取"一节在第 1 章.

__GFP_NOWARN

这个很少用到的标志阻止内核来发出警告(使用 printk), 当一个分配无法满足.

__GFP_HIGH

这个标志标识了一个高优先级请求, 它被允许来消耗甚至被内核保留给紧急状况的最后的内存页.

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY

这些标志修改分配器如何动作, 当它有困难满足一个分配. __GFP_REPEAT 意思是"更尽力些尝试" 通过重复尝试 -- 但是分配可能仍然失败. __GFP_NOFAIL 标志告诉分配器不要失败; 它尽最大努力来满足要求. 使用 __GFP_NOFAIL 是强烈不推荐的; 可能从不会有有效的理由在一个设备驱动中使用它. 最后, __GFP_NORETRY 告知分配器立即放弃如果得不到请求的内存.

8.1.1.1. 内存区

__GFP_DMA 和 __GFP_HIGHMEM 都有一个平台相关的角色, 尽管对所有平台它们的使用都有效.

Linux 内核知道最少 3 个内存区: DMA-能够 内存, 普通内存, 和高端内存. 尽管通常地分配都发生于普通区, 设置这些刚刚提及的位的任一个请求从不同的区来分配内存. 这个想法是, 每个必须知道特殊内存范围(不是认为所有的 RAM 等同)的计算机平台将落入这个抽象中.

DMA-能够 的内存是位于一个优先的地址范围, 外设可以在这里进行 DMA 存取. 在大部分的健全的平台, 所有的内存都在这个区. 在 x86, DMA 区用在 RAM 的前 16 MB, 这里传统的 ISA 设备可以进行 DMA; PCI 设备没有这个限制.

高端内存是一个机制用来允许在 32-位 平台存取(相对地)大量内存. 如果没有首先设置一个特殊的映射这个内存无法直接从内核存取并且通常更难使用. 如果你的驱动使用大量内存, 但是, 如果它能够使用高端内存它将在大系统中工作的更好. 高端内存如何工作以及如何使用它的详情见第 1 章的"高端和低端内存"一节.

无论何时分配一个新页来满足一个内存分配请求, 内核都建立一个能够在搜索中使用的内存区的列表. 如果 __GFP_DMA 指定了, 只有 DMA 区被搜索: 如果在低端没有内存可用, 分配失败. 如果没有特别的标志存取, 普通和 DMA 内存都被搜索; 如果 __GFP_HIGHMEM 设置了, 所有的 3 个区都用来搜索一个空闲的页. (注意, 但是, kmalloc 不能分配高端内存.)

情况在非统一内存存取(NUMA)系统上更加复杂. 作为一个通用的规则, 分配器试图定位进行分配的处理器的本地的内存, 尽管有几个方法来改变这个行为.

内存区后面的机制在 mm/page_alloc.c 中实现, 而内存区的初始化在平台特定的文件中, 常常在 arch 目录树的 mm/init.c. 我们将在第 15 章再次讨论这些主题.

8.1.2. size 参数

内核管理系统的物理内存，这些物理内存只是以页大小的块来使用. 结果是，kmalloc 看来非常不同于一个典型的用户空间 malloc 实现. 一个简单的，面向堆的分配技术可能很快有麻烦；它可能在解决页边界时有困难. 因而，内核使用一个特殊的面向页的分配技术来最好地利用系统 RAM.

Linux 处理内存分配通过创建一套固定大小的内存对象池. 分配请求被这样来处理，进入一个持有足够大的对象的池子并且将整个内存块递交给请求者. 内存管理方案是非常复杂，并且细节通常不是全部设备驱动编写者都感兴趣的.

然而，驱动开发者应当记住的一件事情是，内核只能分配某些预定义的，固定大小的字节数组. 如果你请求一个任意数量内存，你可能得到稍微多于你请求的，至多是 2 倍数量. 同样，程序员应当记住 kmalloc 能够处理的最小分配是 32 或者 64 字节，依赖系统的体系所使用的页大小.

kmalloc 能够分配的内存块的大小有一个上限. 这个限制随着体系和内核配置选项而变化. 如果你的代码是要完全可移植，它不能指望可以分配任何大于 128 KB. 如果你需要多于几个 KB，但是，有个比 kmalloc 更好的方法来获得内存，我们在本章后面描述.

8.2. 后备缓存

一个设备驱动常常以反复分配许多相同大小的对象而结束. 如果内核已经维护了一套相同大小对象的内存池，为什么不增加一些特殊的内存池给这些高容量的对象？实际上，内核确实实现了一个设施来创建这类内存池，它常常被称为一个后备缓存. 设备驱动常常不展示这类的内存行为，它们证明使用一个后备缓存是对的，但是，有例外；在 Linux 2.6 中 USB 和 SCSI 驱动使用缓存.

Linux 内核的缓存管理者有时称为" slab 分配器". 因此，它的功能和类型在 <linux/slab.h> 中声明. slab 分配器实现有一个 kmem_cache_t 类型的缓存；使用一个对 kmem_cache_create 的调用来创建它们：

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,
 size_t offset,
 unsigned long flags,
 void (*constructor)(void *, kmem_cache_t *,
 unsigned long flags), void (*destructor)(void *, kmem_cache_t *, unsigned
long flags));
```

这个函数创建一个新的可以驻留任意数目全部同样大小的内存区的缓存对象，大小由 size 参数指定. name 参数和这个缓存关联并且作为一个在追踪问题时有用的管理信息；通常，它被设置为被缓存的结构类型的名子. 这个缓存保留一个指向 name 的指针，而不是拷贝它，因此驱动应当传递一个指向在静态存储中的名子的指针(常常这个名子只是一个文字字串). 这个名子不能包含空格.

offset 是页内的第一个对象的偏移; 它可被用来确保一个对被分配的对象的特殊对齐, 但是你最可能会使用 0 来请求缺省值. flags 控制如何进行分配并且是下列标志的一个位掩码:

SLAB_NO_REAP

设置这个标志保护缓存在系统查找内存时被削减. 设置这个标志通常是个坏主意; 重要的是避免不必要地限制内存分配器的行动自由.

SLAB_HWCACHE_ALIGN

这个标志需要每个数据对象被对齐到一个缓存行; 实际对齐依赖主机平台的缓存分布. 这个选项可以是一个好的选择, 如果在 SMP 机器上你的缓存包含频繁存取的项. 但是, 用来获得缓存行对齐的填充可以浪费可观的内存量.

SLAB_CACHE_DMA

这个标志要求每个数据对象在 DMA 内存区分配.

还有一套标志用来调试缓存分配; 详情见 mm/slab.c. 但是, 常常地, 在用来开发的系统中, 这些标志通过一个内核配置选项被全局性地设置

函数的 constructor 和 destructor 参数是可选函数(但是可能没有 destructor, 如果没有 constructor); 前者可以用来初始化新分配的对象, 后者可以用来"清理"对象在它们的内存被作为一个整体释放回给系统之前.

构造函数和析构函数会有用, 但是有几个限制你必须记住. 一个构造函数在分配一系列对象的内存时被调用; 因为内存可能持有几个对象, 构造函数可能被多次调用. 你不能假设构造函数作为分配一个对象的一个立即的结果而被调用. 同样地, 析构函数可能在以后某个未知的时间中调用, 不是立刻在一个对象被释放后. 析构函数和构造函数可能或不可能被允许睡眠, 根据它们是否被传递 SLAB_CTOR_ATOMIC 标志(这里 CTOR 是 constructor 的缩写).

为方便, 一个程序员可以使用相同的函数给析构函数和构造函数; slab 分配器常常传递 SLAB_CTOR_CONSTRUCTOR 标志当被调用者是一个构造函数.

一旦一个对象的缓存被创建, 你可以通过调用 kmem_cache_alloc 从它分配对象.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

这里, cache 参数是你之前已经创建的缓存; flags 是你会传递给 kmalloc 的相同, 并且被参考如果 kmem_cache_alloc 需要出去并分配更多内存.

为释放一个对象, 使用 kmem_cache_free:

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

当驱动代码用完这个缓存, 典型地当模块被卸载, 它应当如下释放它的缓存:

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

这个销毁操作只在从这个缓存中分配的所有的对象都已返回给它时才成功. 因此, 一个模块应当检查从 kmem_cache_destroy 的返回值; 一个失败指示某类在模块中的内存泄漏(因为某些对象已被丢失.)

使用后备缓存的一方面益处是内核维护缓冲使用的统计. 这些统计可从 /proc/slabinfo 获得.

8.2.1. 一个基于 Slab 缓存的 scull: scullc

是时候给个例子了. scullc 是一个简化的 scull 模块的版本, 它只实现空设备 -- 永久的内存区. 不象 scull, 它使用 kmalloc, scullc 使用内存缓存. 量子的大小可在编译时和加载时修改, 但是不是在运行时 -- 这可能需要创建一个新内存区, 并且我们不想处理这些不必要的细节.

scullc 使用一个完整的例子, 可用来试验 slab 分配器. 它区别于 scull 只在几行代码. 首先, 我们必须声明我们自己的 slab 缓存:

```
/* declare one cache pointer: use it for all devices */
kmem_cache_t *scullc_cache;
```

slab 缓存的创建以这样的方式处理(在模块加载时):

```
/* scullc_init: create a cache for our quanta */
scullc_cache = kmem_cache_create("scullc", scullc_quantum,
                                 0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* no
ctor/dtor */

if (!scullc_cache)
{
        scullc_cleanup();
        return -ENOMEM;

}
```

这是它如何分配内存量子:

```
/* Allocate a quantum using the memory cache */
if (!dptr->data[s_pos])
{
        dptr->data[s_pos] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);
        if (!dptr->data[s_pos])

                goto nomem;
        memset(dptr->data[s_pos], 0, scullc_quantum);
}
```

还有这些代码行释放内存:

```
for (i = 0; i < qset; i++)
        if (dptr->data[i])
                kmem_cache_free(scullc_cache, dptr->data[i]);
```

最后, 在模块卸载时, 我们不得不返回缓存给系统:

```
/* scullc_cleanup: release the cache of our quanta */
if (scullc_cache)
        kmem_cache_destroy(scullc_cache);
```

从 scull 到 scullc 的主要不同是稍稍的速度提升以及更好的内存使用. 因为量子从一个恰好是合适大小的内存片的池中分配, 它们在内存中的排列是尽可能的密集, 与 scull 量子的相反, 它带来一个不可预测的内存碎片.

8.2.2. 内存池

在内核中有不少地方内存分配不允许失败. 作为一个在这些情况下确保分配的方式, 内核开发者创建了一个已知为内存池(或者是 "mempool")的抽象. 一个内存池真实地只是一类后备缓存, 它尽力一直保持一个空闲内存列表给紧急时使用.

一个内存池有一个类型 mempool_t (在 <linux/mempool.h> 中定义); 你可以使用 mempool_create 创建一个:

```
mempool_t *mempool_create(int min_nr,
 mempool_alloc_t *alloc_fn,
 mempool_free_t *free_fn,
 void *pool_data);
```

min_nr 参数是内存池应当一直保留的最小数量的分配的对象. 实际的分配和释放对象由 alloc_fn 和 free_fn 处理, 它们有这些原型:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

给 mempool_create 最后的参数 (pool_data) 被传递给 alloc_fn 和 free_fn.

如果需要, 你可编写特殊用途的函数来处理 mempool 的内存分配. 常常, 但是, 你只需要使内核 slab 分配器为你处理这个任务. 有 2 个函数 (mempool_alloc_slab 和 mempool_free_slab) 来进行在内存池分配原型和 kmem_cache_alloc 和 kmem_cache_free 之间的感应淬火. 因此, 设置内存池的代码常常看来如此:

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM,mempool_alloc_slab, mempool_free_slab,
cache);
```

一旦已创建了内存池, 可以分配和释放对象,使用:

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

当内存池创建了, 分配函数将被调用足够的次数来创建一个预先分配的对象池. 因此, 对 mempool_alloc 的调用试图从分配函数请求额外的对象; 如果那个分配失败, 一个预先分配的对象(如果有剩下的)被返回. 当一个对象被用 mempool_free 释放, 它保留在池中, 如果对齐预分配的对象数目小于最小量; 否则, 它将被返回给系统.

一个 mempool 可被重新定大小, 使用:

```
int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);
```

这个调用, 如果成功, 调整内存池的大小至少有 new_min_nr 个对象. 如果你不再需要一个内存池, 返回给系统使用:

```
void mempool_destroy(mempool_t *pool);
```

你编写返回所有的分配的对象, 在销毁 mempool 之前, 否则会产生一个内核 oops.

如果你考虑在你的驱动中使用一个 mempool, 请记住一件事: mempools 分配一块内存在一个链表中, 对任何真实的使用是空闲和无用的. 容易使用 mempools 消耗大量的内存. 在几乎每个情况下, 首选的可选项是不使用 mempool 并且代替以简单处理分配失败的可能性. 如果你的驱动有任何方法以不危害到系统完整性的方式来响应一个分配失败, 就这样做. 驱动代码中的 mempools 的使用应当少.

8.3. get_free_page 和其友

如果一个模块需要分配大块的内存, 它常常最好是使用一个面向页的技术. 请求整个页也有其他的优点, 这个在 15 章介绍.

为分配页, 下列函数可用:

get_zeroed_page(unsigned int flags);

返回一个指向新页的指针并且用零填充了该页.

__get_free_page(unsigned int flags);

类似于 get_zeroed_page, 但是没有清零该页.

__get_free_pages(unsigned int flags, unsigned int order);

分配并返回一个指向一个内存区第一个字节的指针, 内存区可能是几个(物理上连续)页长但是没有清零.

flags 参数同 kmalloc 的用法相同; 常常使用 GFP_KERNEL 或者 GFP_ATOMIC, 可能带有 __GFP_DMA 标志(给可能用在 ISA DMA 操作的内存) 或者 __GFP_HIGHMEM 当可能使用

高端内存时. [29]order 是你在请求的或释放的页数的以 2 为底的对数(即, log2N). 例如, 如果你要一个页 order 为 0, 如果你请求 8 页就是 3. 如果 order 太大(没有那个大小的连续区可用), 页分配失败. get_order 函数, 它使用一个整数参数, 可以用来从一个 size 中提取 order(它必须是 2 的幂)给主机平台. order 允许的最大值是 10 或者 11 (对应于 1024 或者 2048 页), 依赖于体系. 但是, 一个 order-10 的分配在除了一个刚刚启动的有很多内存的系统中成功的机会是小的.

如果你好奇, /proc/buddyinfo 告诉你系统中每个内存区中的每个 order 有多少块可用.

当一个程序用完这些页, 它可以使用下列函数之一来释放它们. 第一个函数是一个落回第二个函数的宏:

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果你试图释放和你分配的页数不同的页数, 内存图变乱, 系统在后面时间中有麻烦.

值得强调一下, __get_free_pages 和其他的函数可以在任何时候调用, 遵循我们看到的 kmalloc 的相同规则. 这些函数不能在某些情况下分配内存, 特别当使用 GFP_ATOMIC 时. 因此, 调用这些分配函数的程序必须准备处理分配失败.

尽管 kmalloc(GFP_KERNEL)有时失败当没有可用内存时, 内核尽力满足分配请求. 因此, 容易通过分配太多的内存降低系统的响应. 例如, 你可以通过塞入一个 scull 设备大量数据使计算机关机; 系统开始爬行当它试图换出尽可能多的内存来满足 kmalloc 的请求. 因为每个资源在被增长的设备所吞食, 计算机很快就被说无法用; 在这点上, 你甚至不能启动一个新进程来试图处理这个问题. 我们在 scull 不解释这个问题, 因为它只是一个例子模块并且不是一个真正的放入多用户系统的工具. 作为一个程序员, 你必须小心, 因为一个模块是特权代码并且可能在系统中开启新的安全漏洞(最可能是一个服务拒绝漏洞好像刚刚描述过的.)

8.3.1. 一个使用整页的 scull: scullp

为了真实地测试页分配, 我们已随其他例子代码发布了 scullp 模块. 它是一个简化的 scull, 就像前面介绍过的 scullc.

scullp 分配的内存量子是整页或者页集合: scullp_order 变量缺省是 0, 但是可以在编译或加载时改变.

下列代码行显示了它如何分配内存:

```
/* Here's the allocation of a single quantum */
if (!dptr->data[s_pos])
{
        dptr->data[s_pos] =
```

[29] 尽管 alloc_pages (稍后描述)应当真正地用作分配高端内存页, 由于某些理由我们直到 15 章才真正涉及.

```
                (void *)__get_free_pages(GFP_KERNEL, dptr->order);
        if (!dptr->data[s_pos])
                goto nomem;
        memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

scullp 中释放内存的代码看来如此:

```
/* This code frees a whole quantum-set */
for (i = 0; i < qset; i++)
        if (dptr->data[i])
                free_pages((unsigned long)(dptr->data[i]), dptr->order);
```

在用户级别, 被感觉到的区别主要是一个速度提高和更好的内存使用, 因为没有内部的内存碎片. 我们运行一些测试从 scull0 拷贝 4 MB 到 scull1, 并且接着从 scullp0 到 scullp1; 结果显示了在内核空间处理器使用率有轻微上升.

性能的提高不是激动人心的, 因为 kmalloc 被设计为快的. 页级别分配的主要优势实际上不是速度, 而是更有效的内存使用. 按页分配不浪费内存, 而使用 kmalloc 由于分配的粒度会浪费无法预测数量的内存.

但是 __get_free_page 函数的最大优势是获得的页完全是你的, 并且你可以, 理论上, 可以通过适当的设置页表来组合这些页为一个线性的区域. 例如, 你可以允许一个用户进程 mmap 作为单个不联系的页而获得的内存区. 我们在 15 章讨论这种操作, 那里我们展示 scullp 如何提供内存映射, 一些 scull 无法提供的东西.

8.3.2. alloc_pages 接口

为完整起见, 我们介绍另一个内存分配的接口, 尽管我们不会准备使用它直到 15 章. 现在, 能够说 struct page 是一个描述一个内存页的内部内核结构. 如同我们将见到的, 在内核中有许多地方有必要使用页结构; 它们是特别有用的, 在任何你可能处理高端内存的情况下, 高端内存在内核空间中没有一个常量地址.

Linux 页分配器的真正核心是一个称为 alloc_pages_node 的函数:

```
struct page *alloc_pages_node(int nid, unsigned int flags,
 unsigned int order);
```

这个函数页有 2 个变体(是简单的宏); 它们是你最可能用到的版本:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

核心函数, alloc_pages_node, 使用 3 个参数, nid 是要分配内存的 NUMA 节点 ID[30][30], flags 是通常的 GFP_ 分配标志, 以及 order 是分配的大小. 返回值是一个指向描述分配的内存的第一个(可能许多)页结构的指针, 或者, 如常, NULL 在失败时.

alloc_pages 简化了情况, 通过在当前 NUMA 节点分配内存(它使用 numa_node_id 的返回值作为 nid 参数调用 alloc_pages_node). 并且, 当然, alloc_pages 省略了 order 参数并且分配一个单个页.

为释放这种方式分配的页, 你应当使用下列一个:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

如果你对一个单个页的内容是否可能驻留在处理器缓存中有特殊的认识, 你应当使用 free_hot_page (对于缓存驻留的页) 或者 free_cold_page 通知内核. 这个信息帮助内存分配器在系统中优化它的内存使用.

8.3.3. vmalloc 和 其友

我们展示给你的下一个内存分配函数是 vmlloc, 它在虚拟内存空间分配一块连续的内存区. 尽管这些页在物理内存中不连续 (使用一个单独的对 alloc_page 的调用来获得每个页), 内核看它们作为一个一个连续的地址范围. vmalloc 返回 0 (NULL 地址) 如果发生一个错误, 否则, 它返回一个指向一个大小至少为 size 的连续内存区.

我们这里描述 vmalloc 因为它是一个基本的 Linux 内存分配机制. 我们应当注意, 但是, vmalloc 的使用在大部分情况下不鼓励. 从 vmalloc 获得的内存用起来稍微低效些, 并且, 在某些体系上, 留给 vmalloc 的地址空间的数量相对小. 使用 vmalloc 的代码如果被提交来包含到内核中可能会受到冷遇. 如果可能, 你应当直接使用单个页而不是试图使用 vmalloc 来掩饰事情.

让我们看看 vmalloc 如何工作的. 这个函数的原型和它相关的东西(ioremap, 严格地不是一个分配函数, 在本节后面讨论)是下列:

```
#include <linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

30 [30] NUMA (非统一内存存取) 计算机是多处理器系统, 这里内存对于特定的处理器组("节点")是"局部的". 对局部内存的存取比存取非局部内存更快. 在这样的系统, 在当前节点分配内存是重要的. 驱动作者通常不必担心 NUMA 问题, 但是.

值得强调的是 kmalloc 和 _get_free_pages 返回的内存地址也是虚拟地址. 它们的实际值在它用在寻址物理地址前仍然由 MMU (内存管理单元, 常常是 CPU 的一部分)管理. [31][31] vmalloc 在它如何使用硬件上没有不同, 不同是在内核如何进行分配任务上.

kmalloc 和 __get_free_pages 使用的(虚拟)地址范围特有一个一对一映射到物理内存, 可能移位一个常量 PAGE_OFFSET 值; 这些函数不需要给这个地址范围修改页表. vmalloc 和 ioremap 使用的地址范围, 另一方面, 是完全地合成的, 并且每个分配建立(虚拟)内存区域, 通过适当地设置页表.

这个区别可以通过比较分配函数返回的指针来获知. 在一些平台(例如, x86), vmalloc 返回的地址只是远离 kmalloc 使用的地址. 在其他平台上(例如, MIPS, IA-64, 以及 x86_64), 它们属于一个完全不同的地址范围. 对 vmalloc 可用的地址在从 VMALLOC_START 到 VAMLLOC_END 的范围中. 2 个符号都定义在 <asm/patable.h> 中.

vmalloc 分配的地址不能用于微处理器之外, 因为它们只在处理器的 MMU 之上才有意义. 当一个驱动需要一个真正的物理地址(例如一个 DMA 地址, 被外设硬件用来驱动系统的总线), 你无法轻易使用 vmalloc. 调用 vmalloc 的正确时机是当你在为一个大的只存在于软件中的顺序缓冲分配内存时. 重要的是注意 vamlloc 比 __get_free_pages 有更多开销, 因为它必须获取内存并且建立页表. 因此, 调用 vmalloc 来分配仅仅一页是无意义的.

在内核中使用 vmalloc 的一个例子函数是 create_module 系统调用, 它使用 vmalloc 为在创建的模块获得空间. 模块的代码和数据之后被拷贝到分配的空间中, 使用 copy_from_user. 在这个方式中, 模块看来是加载到连续的内存. 你可以验证, 同过看 /proc/kallsyms, 模块输出的内核符号位于一个不同于内核自身输出的符号的内存范围.

使用 vmalloc 分配的内存由 vfree 释放, 采用和 kfree 释放由 kmalloc 分配的内存的相同方式.

如同 vmalloc, ioremap 建立新页表; 不同于 vmalloc, 但是, 它实际上不分配任何内存. ioremap 的返回值是一个特殊的虚拟地址可用来存取特定的物理地址范围; 获得的虚拟地址应当最终通过调用 iounmap 来释放.

ioremap 对于映射一个 PCI 缓冲的(物理)地址到(虚拟)内核空间是非常有用的. 例如, 它可用来存取一个 PCI 视频设备的帧缓冲; 这样的缓冲常常被映射在高端物理地址, 在内核启动时建立的页表的地址范围之外. PCI 问题在 12 章有详细解释.

由于可移植性, 值得注意的是你不应当直接存取由 ioremap 返回的地址好像是内存指针. 你应当一直使用 readb 和 其他的在第 9 章介绍的 I/O 函数. 这个要求适用因为一些平台, 例如 Alpha, 无法直接映射 PCI 内存区到处理器地址空间, 由于在 PCI 规格和 Alpha 处理器之间的在数据如何传送方面的不同.

[31] [31] 实际上, 一些体系结构定义"虚拟"地址为保留给寻址物理内存. 当这个发生了, Linux 内核利用这个特性, 并且 kernel 和 __get_free_pages 地址都位于这些地址范围中的一个. 这个区别对设备驱动和其他的不直接包含到内存管理内核子系统中的代码是透明的.

ioremap 和 vmalloc 是面向页的(它通过修改页表来工作); 结果, 重分配的或者分配的大小被调整到最近的页边界. ioremap 模拟一个非对齐的映射通过"向下调整"被重映射的地址以及通过返回第一个被重映射页内的偏移.

vmalloc 的一个小的缺点在于它无法在原子上下文中使用, 因为, 内部地, 它使用 kmalloc(GFP_KERNEL) 来获取页表的存储, 并且因此可能睡眠. 这不应当是一个问题 -- 如果 __get_free_page 的使用对于一个中断处理不足够好, 软件设计需要一些清理.

8.3.4. 一个使用虚拟地址的 scull : scullv

使用 vmalloc 的例子代码在 scullv 模块中提供. 如同 scullp, 这个模块是一个 scull 的简化版本, 它使用一个不同的分配函数来为设备存储数据获得空间.

这个模块分配内存一次 16 页. 分配以大块方式进行来获得比 scullp 更好的性能, 并且来展示一些使用其他分配技术要花很长时间的东西是可行的. 使用 __get_free_pages 来分配多于一页是易于失败的, 并且就算它成功了, 它可能是慢的. 如同我们前面见到的, vmalloc 在分配几个页时比其他函数更快, 但是当获取单个页时有些慢, 因为页表建立的开销. scullv 被设计象 scullp 一样. order 指定每个分配的"级数"并且缺省为 4. scullv 和 scullp 之间的位于不同是在分配管理上. 这些代码行使用 vmalloc 来获得新内存:

```
/* Allocate a quantum using virtual addresses */
if (!dptr->data[s_pos])
{
        dptr->data[s_pos] =
                (void *)vmalloc(PAGE_SIZE << dptr->order);
        if (!dptr->data[s_pos])
                goto nomem;
        memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

以及这些代码行释放内存:

```
/* Release the quantum-set */
for (i = 0; i < qset; i++)
        if (dptr->data[i])
                vfree(dptr->data[i]);
```

如果你在使能调试的情况下编译 2 个模块, 你能看到它们的数据分配通过读取它们在 /proc 创建的文件. 这个快照从一套 x86_64 系统上获得:

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135

item at 000001001847da58, qset at 000001001db4c000
0:1001db56000
1:1003d1c7000
```

```
salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem
Device 0: qset 500, order 4, sz 1535135
item at 000001001847da58, qset at 0000010013dea000
0:ffffff0001177000
1:ffffff0001188000
```

下面的输出，相反，来自 x86 系统:

```
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135
item at ccf80e00, qset at cf7b9800
0:ccc58000
1:cccdd000
rudo% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem
Device 0: qset 500, order 4, sz 1535135
item at cfab4800, qset at cf8e4000
0:d087a000
1:d08d2000
```

这些值显示了 2 个不同的行为. 在 x86_64, 物理地址和虚拟地址是完全映射到不同的地址范围(0x100 和 0xffffff00), 而在 x86 计算机上, vmalloc ；虚拟地址只在物理地址使用的映射之上.

8.4. 每-CPU 的变量

每-CPU 变量是一个有趣的 2.6 内核的特性. 当你创建一个每-CPU 变量, 系统中每个处理器获得它自己的这个变量拷贝. 这个可能象一个想做的奇怪的事情, 但是它有自己的优点. 存取每-CPU 变量不需要(几乎)加锁, 因为每个处理器使用它自己的拷贝. 每-CPU 变量也可存在于它们各自的处理器缓存中, 这样对于频繁更新的量子带来了显著的更好性能.

一个每-CPU 变量的好的使用例子可在网络子系统中找到. 内核维护无结尾的计数器来跟踪有每种报文类型有多少被接收; 这些计数器可能每秒几千次地被更新. 不去处理缓存和加锁问题, 网络开发者将统计计数器放进每-CPU 变量. 现在更新是无锁并且快的. 在很少的机会用户空间请求看到计数器的值, 相加每个处理器的版本并且返回总数是一个简单的事情.

每-CPU 变量的声明可在 <linux/percpu.h> 中找到. 为在编译时间创建一个每-CPU 变量, 使用这个宏定义:

```
DEFINE_PER_CPU(type, name);
```

如果这个变量(称为 name 的)是一个数组, 包含这个类型的维数信息. 因此, 一个有 3 个整数的每-CPU 数组应当被创建使用:

```
DEFINE_PER_CPU(int[3], my_percpu_array);
```

每-CPU 变量几乎不必使用明确的加锁来操作. 记住 2.6 内核是可抢占的; 对于一个处理器, 在修改一个每-CPU 变量的临界区中不应当被抢占. 并且如果你的进程在对一个每-CPU 变量存取时将, 要被移动到另一个处理器上, 也不好. 因为这个原因, 你必须显式使用 get_cpu_var 宏来存取当前处理器的给定变量拷贝, 并且当你完成时调用 put_cpu_var. 对 get_cpu_var 的调用返回一个 lvalue 给当前处理器的变量版本并且禁止抢占. 因为一个 lvalue 被返回, 它可被赋值给或者直接操作. 例如, 一个网络代码中的计数器时使用这 2 个语句来递增的:

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

你可以存取另一个处理器的变量拷贝, 使用:

```
per_cpu(variable, int cpu_id);
```

如果你编写使处理器涉及到对方的每-CPU 变量的代码, 你, 当然, 一定要实现一个加锁机制来使存取安全.

动态分配每-CPU 变量也是可能的. 这些变量可被分配, 使用:

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
```

在大部分情况, alloc_percpu 做的不错; 你可以调用 __alloc_percpu 在需要一个特别的对齐的情况下. 在任一情况下, 一个 每-CPU 变量可以使用 free_percpu 被返回给系统. 存取一个动态分配的每-CPU 变量通过 per_cpu_ptr 来完成:

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

这个宏返回一个指针指向 per_cpu_var 对应于给定 cpu_id 的版本. 如果你在简单地读另一个 CPU 的这个变量的版本, 你可以解引用这个指针并且用它来完成. 如果, 但是, 你在操作当前处理器的版本, 你可能需要首先保证你不能被移出那个处理器. 如果你存取这个每-CPU 变量的全部都持有一个自旋锁, 万事大吉. 常常, 但是, 你需要使用 get_cpu 来阻止在使用变量时的抢占. 因此, 使用动态每-CPU 变量的代码会看来如此:

```
int cpu;
cpu = get_cpu()
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* work with ptr */
put_cpu();
```

当使用编译时每-CPU 变量时, get_cpu_var 和 put_cpu_var 宏来照看这些细节. 动态每-CPU 变量需要更多的显式的保护.

每-CPU 变量能够输出给每个模块, 但是你必须使用一个特殊的宏版本:

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

为在一个模块内存取这样一个变量，声明它，使用:

```
DECLARE_PER_CPU(type, name);
```

DECLARE_PER_CPU 的使用(不是 DEFINE_PER_CPU)告知编译器进行一个外部引用.

如果你想使用每-CPU 变量来创建一个简单的整数计数器，看一下在 <linux/percpu_counter.h> 中的现成的实现. 最后，注意一些体系有有限数量的地址空间变量给每-CPU 变量. 如果你创建每-CPU 变量在你自己的代码，你应当尽量使它们小.

8.5. 获得大量缓冲

我们我们已经在前面章节中注意到的，大量连续内存缓冲的分配是容易失败的. 系统内存长时间会碎片化，并且常常出现一个真正的大内存区会完全不可得. 因为常常有办法不使用大缓冲来完成工作，内核开发者没有优先考虑使大分配能工作. 在你试图获得一个大内存区之前，你应当真正考虑一下其他的选择. 到目前止最好的进行大 I/O 操作的方法是通过发散/汇聚操作，我们在第 1 章的"发散-汇聚 映射"一节中讨论了.

8.5.1. 在启动时获得专用的缓冲

如果你真的需要一个大的物理上连续的缓冲，最好的方法是在启动时请求内存来分配它. 在启动时分配是获得连续内存页而避开 __get_free_pages 施加的对缓冲大小限制的唯一方法，不但最大允许大小还有限制的大小选择. 在启动时分配内存是一个"脏"技术，因为它绕开了所有的内存管理策略通过保留一个私有的内存池. 这个技术是不优雅和不灵活的，但是它也是最不易失败的. 不必说，一个模块无法在启动时分配内存；只有直接连接到内核的驱动可以这样做.

启动时分配的一个明显问题是对通常的用户它不是一个灵活的选择，因为这个机制只对连接到内核映象中的代码可用. 一个设备驱动使用这种分配方法可以被安装或者替换只能通过重新建立内核并且重启计算机.

当内核被启动，它赢得对系统种所有可用物理内存的存取. 它接着初始化每个子系统通过调用子系统的初始化函数，允许初始化代码通过减少留给正常系统操作使用的 RAM 数量，来分配一个内存缓冲给自己用.

启动时内存分配通过调用下面一个函数进行:

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

这些函数分配或者整个页(如果它们以 _pages 结尾)或者非页对齐的内存区. 分配的内存可能是高端内存除非使用一个 _low 版本. 如果你在为一个设备驱动分配这个缓冲，你可能想用它做 DMA 操作，并且这对于高端内存不是一直可能的；因此，你可能想使用一个 _low 变体.

很少在启动时释放分配的内存; 你会几乎肯定不能之后取回它, 如果你需要它. 但是, 有一个接口释放这个内存:

```
void free_bootmem(unsigned long addr, unsigned long size);
```

注意以这个方式释放的部分页不返回给系统 -- 但是, 如果你在使用这个技术, 你已可能分配了不少数量的整页来用.

如果你必须使用启动时分配, 你需要直接连接你的驱动到内核. 应当如何完成的更多信息看在内核源码中 Documentation/kbuild 下的文件.

8.6. 快速参考

相关于内存分配的函数和符号是:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

内存分配的最常用接口.

```
#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
```

控制内存分配如何进行的标志, 从最少限制的到最多的. GFP_USER 和 GFP_KERNEL 优先级允许当前进程被置为睡眠来满足请求. GFP_NOFS 和 GFP_NOIO 禁止文件系统操作和所有的 I/O 操作, 分别地, 而 GFP_ATOMIC 分配根本不能睡眠.

```
__GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
```

这些标志修改内核的行为, 当分配内存时.

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset,
unsigned long flags, constructor(), destructor( ));
int kmem_cache_destroy(kmem_cache_t *cache);
```

创建和销毁一个 slab 缓存. 这个缓存可被用来分配几个相同大小的对象.

```
SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
SLAB_CACHE_DMA
```

在创建一个缓存时可指定的标志.

```
SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR
```

分配器可用传递给构造函数和析构函数的标志.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

从缓存中分配和释放一个单个对象. /proc/slabinfo 一个包含对 slab 缓存使用情况统计的虚拟文件.

```
#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
mempool_free_t *free_fn, void *data);
void mempool_destroy(mempool_t *pool);
```

创建内存池的函数, 它试图避免内存分配设备, 通过保持一个已分配项的"紧急列表".

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

从(并且返回它们给)内存池分配项的函数.

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long order);
```

面向页的分配函数. get_zeroed_page 返回一个单个的, 零填充的页. 这个调用的所有的其他版本不初始化返回页的内容.

```
int get_order(unsigned long size);
```

返回关联在当前平台的大小的分配级别, 根据 PAGE_SIZE. 这个参数必须是 2 的幂, 并且返回值至少是 0.

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

释放面向页分配的函数.

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

Linux 内核中最底层页分配器的所有变体.

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
```

使用一个 alloc_page 形式分配的页的各种释放方法.

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

分配或释放一个连续虚拟地址空间的函数. iormap 存取物理内存通过虚拟地址, 而 vmalloc 分配空闲页. 使用 ioreamp 映射的区是 iounmap 释放, 而从 vmalloc 获得的页使用 vfree 来释放.

```
#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
```

定义和声明每-CPU 变量的宏.

```
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
```

提供对静态声明的每-CPU 变量存取的宏.

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
```

进行运行时分配和释放每-CPU 变量的函数.

```
int get_cpu( );
void put_cpu( );
per_cpu_ptr(void *variable, int cpu_id)
```

get_cpu 获得对当前处理器的引用(因此, 阻止抢占和移动到另一个处理器)并且返回处理器的 ID; put_cpu 返回这个引用. 为存取一个动态分配的每-CPU 变量, 用

应当被存取版本所在的 CPU 的 ID 来使用 per_cpu_ptr. 对一个动态的每-CPU 变量当前 CPU 版本的操作, 应当用对 get_cpu 和 put_cpu 的调用来包围.

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
void free_bootmem(unsigned long addr, unsigned long size);
```

在系统启动时进行分配和释放内存的函数(只能被直接连接到内核中去的驱动使用)

第 9 章 与硬件通讯

尽管摆弄 scull 和类似的玩具是对于 Linux 设备驱动的软件接口一个很好的入门, 但是实现一个真正的设备需要硬件. 驱动是软件概念和硬件电路之间的抽象层; 如同这样, 需要与两者沟通. 直到现在, 我们已经检查了软件概念的内部; 本章完成这个图像通过向你展示一个驱动如何存取 I/O 端口和 I/O 内存, 同时在各种 Linux 平台是可移植的.

本章继续尽可能保持独立于特殊的硬件的传统. 但是, 在需要一个特殊例子的地方, 我们使用简单的数字 I/O 端口(例如标准的 PC 并口)来展示 I/O 指令如何工作, 以及正常的帧缓存视频内存来展示内存映射的 I/O.

我们选择简单的数字 I/O, 因为它是一个输入/输出打开的最简单形式. 同样, 并口实现原始 I/O 并且在大部分计算机都有: 写到设备的数据位出现在输出管脚上, 并且处理器可直接存取到输入管脚上的电平. 实际上, 你不得不连接 LED 或者一个打印机到端口上来真正地看到一个数组 I/O 操作的结果, 但是底层硬件非常易于使用.

9.1. I/O 端口和 I/O 内存

每个外设都是通过读写它的寄存器来控制. 大部分时间一个设备有几个寄存器, 并且在连续地址存取它们, 或者在内存地址空间或者在 I/O 地址空间.

在硬件级别上, 内存区和 I/O 区域没有概念上的区别: 它们都是通过在地址总线和控制总线上发出电信号来存取(即, 读写信号)[32][32]并且读自或者写到数据总线.

但是一些 CPU 制造商在他们的芯片上实现了一个单个地址空间, 有人认为外设不同于内存, 因此, 应该有一个分开的地址空间. 一些处理器(最有名的是 x86 家族)有分开的读和写电线给 I/O 端口和特殊的 CPU 指令来存取端口.

因为外设被建立来适合一个外设总线, 并且大部分流行的 I/O 总线成型在个人计算机上, 即便那些没有单独地址空间给 I/O 端口的处理器, 也必须在存取一些特殊设备时伪装读写端口, 常常利用外部的芯片组或者 CPU 核的额外电路. 后一个方法在用在嵌入式应用的小处理器中常见.

由于同样的理由, Linux 在所有它运行的计算机平台上实现了 I/O 端口的概念, 甚至在那些 CPU 实现一个单个地址空间的平台上. 端口存取的实现有时依赖特殊的主机制造和型号(因为不同的型号使用不同的芯片组来映射总线传送到内存地址空间).

即便外设总线有一个单独的地址空间给 I/O 端口, 不是所有的设备映射它们的寄存器到 I/O 端口. 虽然对于 ISA 外设板使用 I/O 端口是普遍的, 大部分 PCI 设备映射寄存器到一个内存地址区. 这种 I/O 内存方法通常是首选的, 因为它不需要使用特殊目的处理器指令; CPU 核存取内存更加有效, 并且编译器当存取内存时有更多自由在寄存器分配和寻址模式的选择上.

[32] [32] 不是所有的计算机平台使用一个读和一个写信号; 有些有不同的方法来寻址外部电路. 这个不同在软件层次是无关的, 但是, 我们将假设全部有读和写来简化讨论.

9.1.1. I/O 寄存器和常规内存

不管硬件寄存器和内存之间的强相似性，存取 I/O 寄存器的程序员必须小心避免被 CPU(或者编译器)优化所戏弄，它可能修改希望的 I/O 行为.

I/O 寄存器和 RAM 的主要不同是 I/O 操作有边际效果，而内存操作没有：一个内存写的唯一效果是存储一个值到一个位置，并且一个内存读返回最近写到那里的值. 因为内存存取速度对 CPU 性能是至关重要的，这种无边际效果的情况已被多种方式优化：值被缓存，并且 读/写指令被重编排.

编译器能够缓存数据值到 CPU 寄存器而不写到内存，并且即便它存储它们，读和写操作都能够在缓冲内存中进行而不接触物理 RAM. 重编排也可能在编译器级别和在硬件级别都发生：常常一个指令序列能够执行得更快，如果它以不同于在程序文本中出现的顺序来执行，例如，为避免在 RISC 流水线中的互锁. 在 CISC 处理器，要花费相当数量时间的操作能够和其他的并发执行，更快的.

当应用于传统内存时(至少在单处理器系统)这些优化是透明和有益的，但是它们可能对正确的 I/O 操作是致命的，因为它们干扰了那些"边际效果"，这是主要的原因为什么一个驱动存取 I/O 寄存器. 处理器无法预见这种情形，一些其他的操作(在一个独立处理器上运行，或者发生在一个 I/O 控制器的事情)依赖内存存取的顺序. 编译器或者 CPU 可能只尽力胜过你并且重编排你请求的操作；结果可能是奇怪的错误而非常难于调试. 因此，一个驱动必须确保没有进行缓冲并且在存取寄存器时没有发生读或写的重编排.

硬件缓冲的问题是最易面对的:底层的硬件已经配置(或者自动地或者通过 Linux 初始化代码)成禁止任何硬件缓冲，当存取 I/O 区时(不管它们是内存还是端口区域).

对编译器优化和硬件重编排的解决方法是安放一个内存屏障在必须以一个特殊顺序对硬件(或者另一个处理器)可见的操作之间. Linux 提供 4 个宏来应对可能的排序需要:

```
#include <linux/kernel.h>
void barrier(void)
```

这个函数告知编译器插入一个内存屏障但是对硬件没有影响. 编译的代码将所有的当前改变的并且驻留在 CPU 寄存器的值存储到内存，并且后来重新读取它们当需要时. 对屏障的调用阻止编译器跨越屏障的优化，而留给硬件自由做它的重编排.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

这些函数插入硬件内存屏障在编译的指令流中；它们的实际实例是平台相关的. 一个 rmb (read memory barrier) 保证任何出现于屏障前的读在执行任何后续读之前完成. wmb 保证写操作中的顺序，并且 mb 指令都保证. 每个这些指令是一个屏障的超集.

read_barrier_depends 是读屏障的一个特殊的，弱些的形式. 而 rmb 阻止所有跨越屏障的读的重编排，read_barrier_depends 只阻止依赖来自其他读的数据的读的重编排. 区别是微小的，并且它不在所有体系中存在. 除非你确切地理解做什么，并且你有理由相信，一个完整的读屏障确实是一个过度地性能开销，你可能应当坚持使用 rmb.

```
void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
```

屏障的这些版本仅当内核为 SMP 系统编译时插入硬件屏障；否则，它们都扩展为一个简单的屏障调用.

在一个设备驱动中一个典型的内存屏障的用法可能有这样的形式:

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

在这种情况，是重要的，确保所有的控制一个特殊操作的设备寄存器在告诉它开始前已被正确设置. 内存屏障强制写以需要的顺序完成.

因为内存屏障影响性能，它们应当只用在确实需要它们的地方. 屏障的不同类型也有不同的性能特性，因此值得使用最特定的可能类型. 例如，在 x86 体系上，wmb() 目前什么都不做，因为写到处理器外不被重编排. 但是，读被重编排，因此 mb() 被 wmb() 慢.

值得注意大部分的其他的处理同步的内核原语，例如自旋锁和原子的 _t 操作，如同内存屏障一样是函数. 还值得注意的是一些外设总线(例如 PCI 总线)有它们自己的缓冲问题；我们在以后章节遇到时讨论它们.

一些体系允许一个赋值和一个内存屏障的有效组合. 内核提供了几个宏来完成这个组合；在缺省情况下，它们如下定义:

```
#define set_mb(var, value) do {var = value; mb();}  while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

在合适的地方，<asm/system.h> 定义这些宏来使用体系特定的指令来很快完成任务. 注意 set_rmb 只在少量体系上定义. (一个 do...while 结构的使用是一个标准 C 用语，来使被扩展的宏作为一个正常的 C 语句可在所有上下文中工作).

9.2. 使用 I/O 端口

I/O 端口是驱动用来和很多设备通讯的方法，至少部分时间. 这节涉及可用的各种函数来使用 I/O 端口；我们也触及一些可移植性问题.

9.2.1. I/O 端口分配

如同你可能希望的，你不应当离开并开始抨击 I/O 端口而没有首先确认你对这些端口有唯一的权限. 内核提供了一个注册接口以允许你的驱动来声明它需要的端口. 这个接口中的核心的函数是 request_region:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const
char *name);
```

这个函数告诉内核，你要使用 n 个端口，从 first 开始. name 参数应当是你的设备的名子. 如果分配成功返回值是非 NULL. 如果你从 request_region 得到 NULL，你将无法使用需要的端口.

所有的的端口分配显示在 /proc/ioports 中. 如果你不能分配一个需要的端口组，这是地方来看看谁先到那里了.

当你用完一组 I/O 端口(在模块卸载时，也许)，应当返回它们给系统，使用:

```
void release_region(unsigned long start, unsigned long n);
```

还有一个函数以允许你的驱动来检查是否一个给定的 I/O 端口组可用:

```
int check_region(unsigned long first, unsigned long n);
```

这里，如果给定的端口不可用，返回值是一个负错误码. 这个函数是不推荐的，因为它的返回值不保证是否一个分配会成功；检查和后来的分配不是一个原子的操作. 我们列在这里因为几个驱动仍然在使用它，但是你调用一直使用 request_region，它进行要求的加锁来保证分配以一个安全的原子的方式完成.

9.2.2. 操作 I/O 端口

在驱动硬件请求了在它的活动中需要使用的 I/O 端口范围之后，它必须读且/或写到这些端口. 为此，大部分硬件区别 8-位，16-位，和 32-位端口. 常常你无法混合它们，象你正常使用系统内存存取一样.[33] [33]

一个 C 程序，因此，必须调用不同的函数来存取不同大小的端口. 如果在前一节中建议的，只支持唯一内存映射 I/O 寄存器的计算机体系伪装端口 I/O ，通过重新映射端口地址到内存地址，并且内核向驱动隐藏了细节以便易于移植. Linux 内核头文件(特别地，体系依赖的头文件 <asm/io.h>) 定义了下列内联函数来存取 I/O 端口:

33 [33] 有时 I/O 端口象内存一样安排，并且你可(例如)绑定 2 个 8-位 写为一个单个 16-位 操作. 例如，这应用于 PC 视频板. 但是通常，你不能指望这个特色.

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

读或写字节端口(8 位宽). port 参数定义为 unsigned long 在某些平台以及 unsigned short 在其他的上. inb 的返回类型也是跨体系而不同的.

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

这些函数存取 16-位 端口(一个字宽); 在为 S390 平台编译时它们不可用, 它只支持字节 I/O.

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

这些函数存取 32-位 端口. longword 声明为或者 unsigned long 或者 unsigned int, 根据平台. 如同字 I/O, "Long" I/O 在 S390 上不可用.

从现在开始, 当我们使用 unsigned 没有进一步类型规定时, 我们指的是一个体系相关的定义, 它的确切特性是不相关的. 函数几乎一直是可移植的, 因为编译器自动转换值在赋值时 -- 它们是 unsigned 有助于阻止编译时的警告. 这样的转换不丢失信息, 只要程序员安排明智的值来避免溢出. 我们坚持这个"未完成的类型"传统贯串本章.

注意, 没有定义 64-位 端口 I/O 操作. 甚至在 64-位 体系中, 端口地址空间使用一个 32-位(最大)的数据通路.

9.2.3. 从用户空间的 I/O 存取

刚刚描述的这些函数主要打算被设备驱动使用, 但它们也可从用户空间使用, 至少在 PC-类 的计算机. GNU C 库在 <sys/io.h> 中定义它们. 下列条件应当应用来对于 inb 及其友在用户空间代码中使用:

- 程序必须使用 -O 选项编译来强制扩展内联函数.
- ioperm 和 iopl 系统调用必须用来获得权限来进行对端口的 I/O 操作. ioperm 为单独端口获取许可, 而 iopl 为整个 I/O 空间获取许可. 这 2 个函数都是 x86 特有的.
- 程序必须作为 root 来调用 ioperm 或者 iopl. [34][34] 可选地, 一个它的祖先必须已赢得作为 root 运行的端口权限.

如果主机平台没有 ioperm 和 iopl 系统调用, 用户空间仍然可以存取 I/O 端口, 通过使用 /dev/prot 设备文件. 注意, 但是, 这个文件的含义是非常平台特定的, 并且对任何东西除了 PC 不可能有用.

[34] [34] 技术上, 它必须有 CAP_SYS_RAWIO 能力, 但是在大部分当前系统中这是与作为 root 运行是同样的.

例子源码 misc-progs/inp.c 和 misc-progs/outp.c 是一个从命令行读写端口的小工具，在用户空间. 它们希望被安装在多个名子下(例如, inb, inw, 和 inl 并且操作字节, 字, 或者长端口依赖于用户调用哪个名子). 它们使用 ioperm 或者 iopl 在 x86 下, 在其他平台是 /dev/port.

程序可以做成 setuid root, 如果你想过危险生活并且在不要求明确的权限的情况下使用你的硬件. 但是, 请不要在产品系统上以 set-uid 安装它们; 它们是设计上的安全漏洞.

9.2.4. 字串操作

除了单发地输入和输出操作, 一些处理器实现了特殊的指令来传送一系列字节, 字, 或者长字 到和自一个单个 I/O 端口或者同样大小. 这是所谓的字串指令, 并且它们完成任务比一个 C 语言循环能做的更快. 下列宏定义实现字串处理的概念或者通过使用一个单个机器指令或者通过执行一个紧凑的循环, 如果目标处理器没有进行字串 I/O 的指令. 当编译为 S390 平台时这些宏定义根本不定义. 这应当不是个移植性问题, 因为这个平台通常不与其他平台共享设备驱动, 因为它的外设总线是不同的.

字串函数的原型是:

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

读或写从内存地址 addr 开始的 count 字节. 数据读自或者写入单个 port 端口.

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

读或写 16-位 值到一个单个 16-位 端口.

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

读或写 32-位 值到一个单个 32-位 端口.

有件事要记住, 当使用字串函数时: 它们移动一个整齐的字节流到或自端口. 当端口和主系统有不同的字节对齐规则, 结果可能是令人惊讶的. 使用 inw 读取一个端口交换这些字节, 如果需要, 来使读取的值匹配主机字节序. 字串函数, 相反, 不进行这个交换.

9.2.5. 暂停 I/O

一些平台 - 最有名的 i386 - 可能有问题当处理器试图太快传送数据到或自总线. 当处理器对于外设总线被过度锁定时可能引起问题(想一下 ISA)并且可能当设备单板太慢时表现出来. 解决方法是插入一个小的延时在每个 I/O 指令后面, 如果跟随着另一个指令. 在 x86 上, 这个暂停是通过进行一个 outb 指令到端口 0x80 (正常地不是常常用到)实现的, 或者通过忙等待. 细节见你的平台的 asm 子目录的 io.h 文件.

如果你的设备丢失一些数据，或者如果你担心它可能丢失一些，你可以使用暂停函数代替正常的那些. 暂停函数正如前面列出的，但是它们的名子以 _p 结尾；它们称为 inb_p, outb_p, 等等. 这些函数定义给大部分被支持的体系，尽管它们常常扩展为与非暂停 I/O 同样的代码，因为没有必要额外暂停，如果体系使用一个合理的现代外设总线.

9.2.6. 平台依赖性

I/O 指令，由于它们的特性，是高度处理器依赖的. 因为它们使用处理器如何处理移进移出的细节，是非常难以隐藏系统间的不同. 作为一个结果，大部分的关于端口 I/O 的源码是平台依赖的.

你可以看到一个不兼容，数据类型，通过回看函数的列表，这里参数是不同的类型，基于平台间的体系不同点. 例如，一个端口是 unsigned int 在 x86 (这里处理器支持一个 64-KB I/O 空间)，但是在别的平台是 unsiged long，这里的端口只是同内存一样的同一个地址空间中的特殊位置.

其他的平台依赖性来自处理器中的基本的结构性不同，并且，因此，无可避免地. 我们不会进入这个依赖性的细节，因为我们假定你不会给一个特殊的系统编写设备驱动而没有理解底层的硬件. 相反，这是一个内核支持的体系的能力的概括：

IA-32 (x86)
x86_64

> 这个体系支持所有的本章描述的函数. 端口号是 unsigned short 类型.

IA-64 (Itanium)

> 支持所有函数；端口是 unsigned long(以及内存映射的)). 字串函数用 C 实现.

Alpha

> 支持所有函数，并且端口是内存映射的. 端口 I/O 的实现在不同 Alpha 平台上是不同的，根据它们使用的芯片组. 字串函数用 C 实现并且定义在 arch/alpha/lib/io.c 中定义. 端口是 unsigned long.

ARM

> 端口是内存映射的，并且支持所有函数；字串函数用 C 实现. 端口是 unsigned int 类型.

Cris

> 这个体系不支持 I/O 端口抽象，甚至在一个模拟模式；各种端口操作定义成什么不做.

M68k
M68k

端口是内存映射的. 支持字串函数, 并且端口类型是 unsigned char.

MIPS
MIPS64

MIPS 端口支持所有的函数. 字串操作使用紧凑汇编循环来实现, 因为处理器缺乏机器级别的字串 I/O. 端口是内存映射的; 它们是 unsigned long.

PA

支持所有函数; 端口是 int 在基于 PCI 的系统上以及 unsigned short 在 EISA 系统, 除了字串操作, 它们使用 unsigned long 端口号.

PowerPC
PowerPC64

支持所有函数; 端口有 unsigned char * 类型在 32-位 系统上并且 unsigned long 在 64-位 系统上.

S390

类似于 M68k, 这个平台的头文件只支持字节宽的端口 I/O, 而没有字串操作. 端口是 char 指针并且是内存映射的.

Super

端口是 unsigned int (内存映射的), 并且支持所有函数.

SPARC SPARC64

再一次, I/O 空间是内存映射的. 端口函数的版本定义来使用 unsigned long 端口.

好奇的读者能够从 io.h 文件中获得更多信息, 这个文件有时定义几个结构特定的函数, 加上我们在本章中描述的那些. 但是, 警告有些这些文件是相当难读的.

有趣的是注意没有 x86 家族之外的处理器具备一个不同的地址空间给端口, 尽管几个被支持的家族配备有 ISA 和/或 PCI 插槽 (并且 2 种总线实现分开的 I/O 和地址空间).

更多地, 有些处理器(最有名的是早期的 Alphas)缺乏一次移动一个或 2 个字节的指令.[35][35] 因此, 它们的外设芯片组模拟 8-位 和 16-位 I/O 存取, 通过映射它们到内存地址空间的特殊的地址范围. 因此, 操作同一个端口的一个 inb 和 一个 inw 指令, 通过 2 个操作不同地址的 32-位内存读来实现. 幸运的是, 所有这些都对设备驱动编写者隐藏了, 通过本节中描述的宏的内部, 但是我们觉得它是一个要注意的有趣的特性. 如果你想深入探究, 查找在 include/asm-alpha/core_lca.h 中的例子.

[35] [35] 单字节 I/O 不是一个人可能想象的那么重要, 因为它是一个稀少的操作. 为读/写一个单字节到任何地址空间, 你需要实现一个数据通道, 连接寄存器组的数据总线的低位到外部数据总线的任意字节位置. 这些数据通道需要额外的逻辑门在每个数据传输的通道上. 丢掉字节宽的载入和存储能够使整个系统性能受益.

在每个平台的程序员手册中充分描述了 I/O 操作如何在每个平台上进行; 这些手册常常在 WEB 上作为 PDF 下载.

9.3. 一个 I/O 端口例子

我们用来展示一个设备驱动内的端口 I/O 的例子代码, 操作通用的数字 I/O 端口; 这样的端口在大部分计算机系统中找到.

一个数字 I/O 端口, 在它的大部分的普通的化身中, 是一个字节宽的 I/O 位置, 或者内存映射的或者端口映射的. 当你写一个值到一个输出位置, 在输出管脚上见到的电信号根据写入的单个位而改变. 当你从一个输入位置读取一个值, 输入管脚上所见的当前逻辑电平作为单个位的值被返回.

这样的 I/O 端口的实际实现和软件接口各个系统不同. 大部分时间, I/O 管脚由 2 个 I/O 位置控制: 一个允许选择使用那些位作为输入, 哪些位作为输出, 以及一个可以实际读或写逻辑电平的. 有时, 但是, 事情可能更简单, 并且这些位是硬连线为输入或输出(但是, 在这个情况下, 它们不再是所谓的"通用 I/O"); 在所有个人计算机上出现的并口是这样一个非通用 I/O 端口. 任一方式, I/O 管脚对我们马上介绍的例子代码是可用的.

9.3.1. 并口纵览

因为我们期望大部分读者以所谓的"个人计算机"的形式使用一个 x86 平台, 我们觉得值得解释一下 PC 并口如何设计的. 并口是在个人计算机上运行数字 I/O 例子代码的外设接口选择. 尽管大部分读者可能有并口规范用, 为你的方便, 我们在这里总结一下它们.

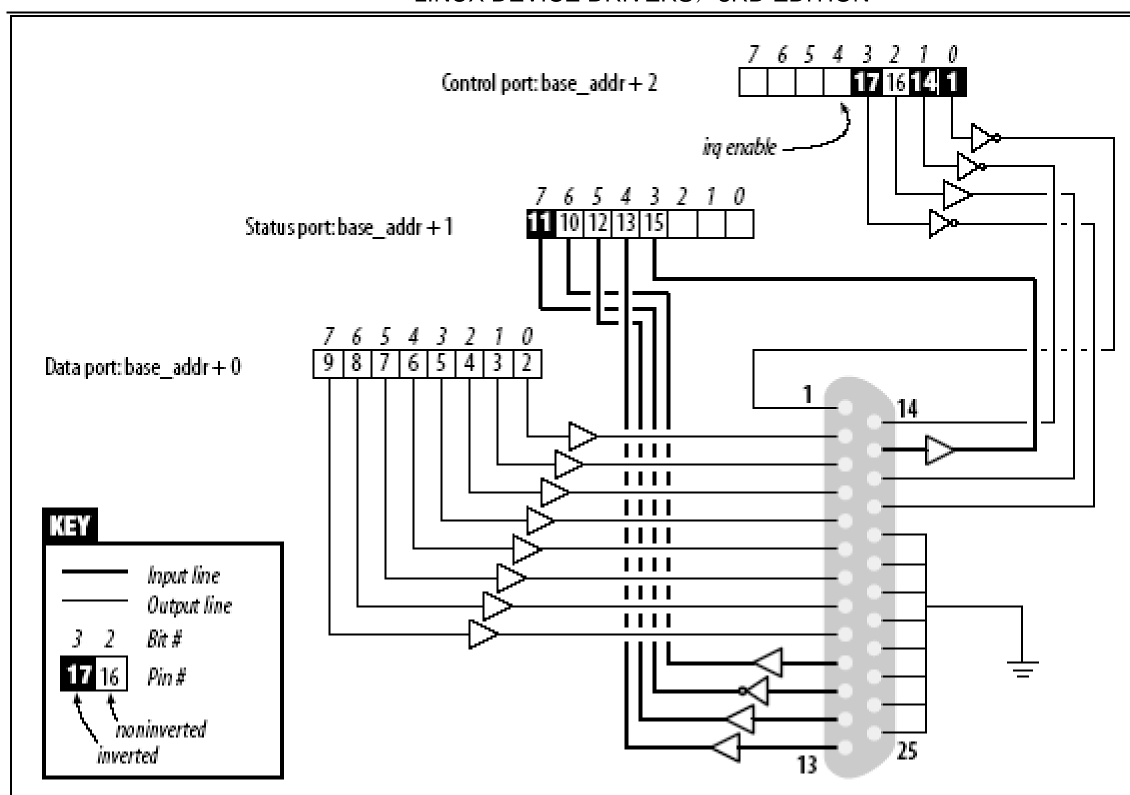
并口, 在它的最小配置中 (我们浏览一下 ECP 和 EPP 模式) 由 3 个 8-位端口组成. PC 标准在 0x378 开始第一个并口的 I/O 端口并且第 2 个在 0x278. 第一个端口是一个双向数据寄存器; 它直接连接到物理连接器的管脚 2 - 9. 第 2 个端口是一个只读状态寄存器; 当并口为打印机使用, 这个寄存器报告打印机状态的几个方面, 例如正在线, 缺纸, 或者忙. 第 3 个端口是一个只出控制寄存器, 它, 在其他东西中, 控制是否中断使能.

并口通讯中使用的信号电平是标准的 TTL 电平: 0 和 5 伏特, 逻辑门限在大概 1.2 伏特. 你可依靠端口至少符合标准 TTL LS 电流规格, 尽管大部分现代并口在电流和电压额定值都工作的好.

并口连接器和计算机内部电路不隔离, 当你想直接连接逻辑门到这个端口是有用的. 但是你不得不小心地正确连接线; 并口电路当你使用你自己的定制电路时容易损坏, 除非你给你的电路增加绝缘. 你可以选择使用插座并口如果你害怕会损坏你的主板.

位的规范在图 并口的管脚 中概述. 你可以存取 12 个输出位和 5 个输入位, 有些是在它们地信号路径上逻辑地翻转了. 唯一的没有关联信号管脚的位是端口 2 的位 4 (0x10), 它使能来自并口的中断. 我们使用这个位作为我们的在第 10 章中的中断处理的实现的一部分.

图 9.1. 并口的管脚



9.3.2. 一个例子驱动

我们介绍的驱动称为 short (Simple Hardware Operations and Raw Tests). 所有它做的是读和写几个 8-位 端口, 从你在加载时选择的开始. 缺省地, 它使用分配给 PC 并口的端口范围. 每个设备节点(有一个独特的次编号)存取一个不同的端口. short 驱动不做任何有用的事情; 它只是隔离来作为操作端口的单个指令给外部使用. 如果你习惯端口 I/O, 你可以使用 short 来熟悉它; 你能够测量它花费来通过端口传送数据的时间或者其他游戏的时间.

为 short 在你的系统上运行, 必须有存取底层硬件设备的自由(缺省地, 并口); 因此, 不能有其他驱动已经分配了它. 大部分现代发布设置并口驱动作为只在需要时加载的模块, 因此对 I/O 地址的竞争常常不是个问题. 如果, 但是, 你从 short 得到一个"无法获得 I/O 地址" 错误(在控制台上或者在系统 log 文件), 一些其他的驱动可能已经获得这个端口. 一个快速浏览 /proc/ioports 常常告诉你哪个驱动在捣乱. 同样的告诫应用于另外 I/O 设备如果你没有在使用并口.

从现在开始, 我们只是用"并口"来简化讨论. 但是, 你能够设置基本的模块参数在加载时来重定向 short 到其他 I/O 设备. 这个特性允许例子代码在任何 Linux 平台上运行, 这里你对一个数字 I/O 接口有权限通过 outb 和 inb 存取(尽管实际的硬件是内存映射的, 除 x86 外的所有平台). 后面, 在"使用 I/O 内存"的一节, 我们展示 short 如何用来使用通用的内存映射数字 I/O.

为观察在并口上发生了什么以及如果你有使用硬件的爱好, 你可以焊接尽管 LED 到输出管脚. 每个 LED 应当串连一个 1-K 电阻导向一个地引脚(除非, 当然, 你的 LED 有内嵌的电阻). 如果你连接一个输出引脚到一个输入管脚, 你会产生你自己的输入能够从输入端口读到.

注意，你无法只连接一个打印机到并口并且看到数据发向 short. 这个驱动实现简单的对 I/O 端口的存取，并且没有进行与打印机需要的来操作数据的握手；在下一章，我们展示了一个例子驱动(称为 shortprint)，它能够驱动并口打印机；这个驱动使用中断，但是，因此我们还是不能到这一点.

如果你要查看并口数据通过焊接 LED 到一个 D-型 连接器，我们建议你不要使用管脚 9 和管脚 10，因为我们之后连接它们在一起来运行第 10 章展示的例子代码.

只考虑到 short，/dev/short0 写到和读自位于 I/O 基地址的 8-bit 端口(0x378，除非在加载时间改变). /dev/short1 写到位于基址 + 1 的 8-位，等等直到基址 + 7.

/dev/short0 进行的实际输出操作是基于使用 outb 的一个紧凑循环. 一个内存屏障指令用来保证输出操作实际发生并且不被优化掉:

```
while (count--) {
 outb(*(ptr++), port);
    wmb();
}
```

你可以运行下列命令来点亮你的 LED:

```
echo -n "any string" > /dev/short0
```

每个 LED 监视一个单个的输出端口位. 记住只有最后写入的字符，保持稳定在输出管脚上足够长时间你的眼睛能感觉到. 因此，我们建议你阻止自动插入一个结尾新行，通过传递一个 -n 选项给 echo.

读是通过一个类似的函数，围绕 inb 而不是 outb 建立的. 为了从并口读"有意义的"值，你需要某个硬件连接到连接器的输入管脚来产生信号. 如果没有信号，你会读到一个相同字节的无结尾的流. 如果你选择从一个输出端口读取，你极可能得到写到端口的最后的值(这适用于并口和普通使用的其他数字 I/O 电路). 因此，那些不喜欢拿出他们的烙铁的人可以读取当前的输出值在端口 0x378，通过运行这样一个命令:

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

为演示所有 I/O 指令的使用，每个 short 设备有 3 个变形: /dev/short0 进行刚刚展示的循环，/dev/short0p 使用 outb_p 和 inb_p 代替"快速"函数，并且 /dev/short0s 使用字串指令. 有 8 个这样的设备，从 short0 到 short7. 尽管 PC 并口只有 3 个端口，你可能需要它们更多如果使用不同的 I/O 设备来运行你的测试.

short 驱动进行一个非常少的硬件控制，但是足够来展示如何使用 I/O 端口指令. 感兴趣的读者可能想看看 parpor 和 parport_pc 模块的源码，来知道这个设备在真实生活中能有多复杂来支持一系列并口上的设备(打印机，磁带备份，网络接口)

9.4. 使用 I/O 内存

尽管 I/O 端口在 x86 世界中流行, 用来和设备通讯的主要机制是通过内存映射的寄存器和设备内存. 2 者都称为 I/O 内存, 因为寄存器和内存之间的区别对软件是透明的.

I/O 内存是简单的一个象 RAM 的区域, 它被处理器用来跨过总线存取设备. 这个内存可用作几个目的, 例如持有视频数据或者以太网报文, 同时实现设备寄存器就象 I/O 端口一样的行为(即, 它们有读和写它们相关联的边际效果).

存取 I/O 内存的方式依赖计算机体系, 总线, 和使用的设备, 尽管外设到处都一样. 本章的讨论主要触及 ISA 和 PCI 内存, 而也试图传递通用的信息. 尽管存取 PCI 内存在这里介绍, 一个 PCI 的通透介绍安排在第 12 章.

依赖计算机平台和使用的总线, I/O 内存可以或者不可以通过页表来存取. 当通过页表存取, 内核必须首先安排从你的驱动可见的物理地址, 并且这常常意味着你必须调用 ioremap 在做任何 I/O 之前. 如果不需要页表, I/O 内存位置看来很象 I/O 端口, 并且你只可以使用正确的包装函数读和写它们.

不管是否需要 ioremap 来存取 I/O 内存, 不鼓励直接使用 I/O 内存的指针. 尽管(如同在 "I/O 端口和 I/O 内存" 一节中介绍的)I/O 内存如同在硬件级别的正常 RAM 一样寻址, 在"I/O 寄存器和传统内存"一节中概述的额外的小心建议避免正常的指针. 用来存取 I/O 内存的包装函数在所有平台上是安全的并且在任何时候直接的指针解引用能够进行操作时, 会被优化掉.

因此, 尽管在 x86 上解引用一个指针能工作(在现在), 不使用正确的宏定义阻碍了驱动的移植性和可读性.

9.4.1. I/O 内存分配和映射

I/O 内存区必须在使用前分配. 分配内存区的接口是(在 <linux/ioport.h> 定义):

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
char *name);
```

这个函数分配一个 len 字节的内存区, 从 start 开始. 如果一切顺利, 一个 非NULL 指针返回; 否则返回值是 NULL. 所有的 I/O 内存分配来 /proc/iomem 中列出.

内存区在不再需要时应当释放:

```
void release_mem_region(unsigned long start, unsigned long len);
```

还有一个旧的检查 I/O 内存区可用性的函数:

```
int check_mem_region(unsigned long start, unsigned long len);
```

但是, 对于 check_region, 这个函数是不安全和应当避免的.

在存取内存之前, 分配 I/O 内嵌不是唯一的要求的步骤. 你必须也保证这个 I/O 内存已经对内核是可存取的. 使用 I/O 内存不只是解引用一个指针的事情; 在许多系统, I/O 内存根本不是可以这种方式直接存取的. 因此必须首先设置一个映射. 这是 ioremap 函

数的功能, 在第 1 章的 "vmalloc 及其友"一节中介绍的. 这个函数设计来特别的安排虚拟地址给 I/O 内存区.

一旦装备了 ioremap (和 iounmap), 一个设备驱动可以存取任何 I/O 内存地址, 不管是否它是直接映射到虚拟地址空间. 记住, 但是, 从 ioremap 返回的地址不应当直接解引用; 相反, 应当使用内核提供的存取函数. 在我们进入这些函数之前, 我们最好回顾一下 ioremap 原型和介绍几个我们在前一章略过的细节.

这些函数根据下列定义调用:

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

首先, 你注意新函数 ioremap_nacache. 我们在第 8 章没有涉及它, 因为它的意思是明确地硬件相关的. 引用自一个内核头文件:"It’s useful if some control registers are in such an area, and write combining or read caching is not desirable.". 实际上, 函数实现在大部分计算机平台上与 ioremap 一致: 在所有 I/O 内存已经通过非缓冲地址可见的地方, 没有理由使用一个分开的, 非缓冲 ioremap 版本.

9.4.2. 存取 I/O 内存

在一些平台上, 你可能逃过作为一个指针使用 ioremap 的返回值的惩罚. 这样的使用不是可移植的, 并且, 更加地, 内核开发者已经努力来消除任何这样的使用. 使用 I/O 内存的正确方式是通过一系列为此而提供的函数(通过 <asm/io.h> 定义的).

从 I/O 内存读, 使用下列之一:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

这里, addr 应当是从 ioremap 获得的地址(也许与一个整型偏移); 返回值是从给定 I/O 内存读取的.

有类似的一系列函数来写 I/O 内存:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

如果你必须读和写一系列值到一个给定的 I/O 内存地址, 你可以使用这些函数的重复版本:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
```

```
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

这些函数读或写 count 值从给定的 buf 到 给定的 addr. 注意 count 表达为在被写入的数据大小; ioread32_rep 读取 count 32-位值从 buf 开始.

上面描述的函数进行所有的 I/O 到给定的 addr. 如果, 相反, 你需要操作一块 I/O 地址, 你可使用下列之一:

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

这些函数行为如同它们的 C 库类似物.

如果你通览内核源码, 你可看到许多调用旧的一套函数, 当使用 I/O 内存时. 这些函数仍然可以工作, 但是它们在新代码中的使用不鼓励. 除了别的外, 它们较少安全因为它们不进行同样的类型检查. 但是, 我们在这里描述它们:

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

这些宏定义用来从 I/O 内存获取 8-位, 16-位, 和 32-位 数据值.

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

如同前面的函数, 这些函数(宏)用来写 8-位, 16-位, 和 32-位数据项.

一些 64-位平台也提供 readq 和 writeq, 为 PCI 总线上的 4-字(8-字节)内存操作. 这个 4-字 的命名是一个从所有的真实处理器有 16-位 字的时候的历史遗留. 实际上, 用作 32-位 值的 L 命名也已变得不正确, 但是命名任何东西可能使事情更混淆.

9.4.3. 作为 I/O 内存的端口

一些硬件有一个有趣的特性: 一些版本使用 I/O 端口, 而其他的使用 I/O 内存. 输出给处理器的寄存器在任一种情况中相同, 但是存取方法是不同的. 作为一个使驱动处理这类硬件的生活容易些的方式, 并且作为一个使 I/O 端口和内存存取的区别最小化的方法, 2.6 内核提供了一个函数, 称为 ioport_map:

```
void *ioport_map(unsigned long port, unsigned int count);
```

这个函数重映射 count I/O 端口和使它们出现为 I/O 内存. 从这点以后, 驱动可以在返回的地址上使用 ioread8 和其友并且根本忘记它在使用 I/O 端口.

这个映射应当在它不再被使用时恢复:

```
void ioport_unmap(void *addr);
```

这些函数使 I/O 端口看来象内存. 但是, 注意 I/O 端口必须仍然使用 request_region 在它们以这种方式被重映射前分配.

9.4.4. 重用 short 为 I/O 内存

short 例子模块, 在存取 I/O 端口前介绍的, 也能用来存取 I/O 内存. 为此, 你必须告诉它使用 I/O 内存在加载时; 还有, 你需要改变基地址来使它指向你的 I/O 区.

例如, 这是我们如何使用 short 来点亮调试 LED, 在一个 MIPS 开发板上:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

使用 short 给 I/O 内存是与它用在 I/O 端口上同样的.

下列片段显示了 short 在写入一个内存位置时用的循环:

```
while (count--) {
 iowrite8(*ptr++, address);
    wmb();
}
```

注意, 这里使用一个写内存屏障. 因为在很多体系上 iowrites8 可能转变为一个直接赋值, 需要内存屏障来保证以希望的顺序来发生.

short 使用 inb 和 outb 来显示它如何完成. 对于读者它可能是一个直接的练习, 但是, 改变 short 来使用 ioport_map 重映射 I/O 端口, 并且相当地简化剩下的代码.

9.4.5. 在 1 MB 之下的 ISA 内存

一个最著名的 I/O 内存区是在个人计算机上的 ISA 范围. 这是在 640 KB(0xA0000)和 1 MB(0x100000)之间的内存范围. 因此, 它正好出现于常规内存 RAM 中间. 这个位置可能看起来有点奇怪; 它是一个在 1980 年代早期所作的决定的产物, 当时 640 KB 内存看来多于任何人可能用到的大小.

这个内存方法属于非直接映射的内存类别. [36]你可以读/写几个字节在这个内存范围, 如同前面解释的使用 short 模块, 就是, 通过在加载时设置 use_mem.

尽管 ISA I/O 内存只在 x86-类 计算机中存在, 我们认为值得用几句话和一个例子驱动.

我们不会谈论 PCI 在本章，因为它是最干净的一类 I/O 内存：一旦你知道内存地址，你可简单地重映射和存取它. PCI I/O 内存的"问题"是它不能为本章提供一个能工作的例子，因为我们不能事先知道你的 PCI 内存映射到的物理地址，或者是否它是安全的来存取任一这些范围. 我们选择来描述 ISA 内存范围，因为它不但少干净并且更适合运行例子代码.

为演示存取 ISA 内存，我们还使用另一个 silly 小模块(例子源码的一部分). 实际上，这个称为 silly，作为 Simple Tool for Unloading and Printing ISA Data 的缩写，或者如此的东东.

模块补充了 short 的功能，通过存取整个 384-KB 内存空间和通过显示所有的不同 I/O 功能. 它特有 4 个设备节点来进行同样的任务，使用不同的数据传输函数. silly 设备作为一个 I/O 内存上的窗口，以类似 /dev/mem 的方式. 你可以读和写数据，并且 lseek 到一个任意 I/O 内存地址.

因为 silly 提供了对 ISA 内存的存取，它必须开始于从映射物理 ISA 地址到内核虚拟地址. 在 Linux 内核的早期，一个人可以简单地安排一个指针给一个感兴趣的 ISA 地址，接着直接对它解引用. 在现代世界，但是，我们必须首先使用虚拟内存系统和重映射内存范围. 这个映射使用 ioremap 完成，如同前面为 short 解释的:

```
#define ISA_BASE 0xA0000
#define ISA_MAX 0x100000 /* for general memory access */

/* this line appears in silly_init */
io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap 返回一个指针值，它能被用来使用 ioread8 和其他函数，在"存取 I/O 内存"一节中解释.

让我们回顾我们的例子模块来看看这些函数如何被使用. /dev/sillyb，特有次编号 0，存取 I/O 内存使用 ioread8 和 iowrite8. 下列代码显示了读的实现，它使地址范围 0xA0000-0xFFFF 作为一个虚拟文件在范围 0-0x5FFF. 读函数构造为一个 switch 语句在不同存取模式上；这是 sillyb 例子:

```
case M_8:
 while (count) {
 *ptr = ioread8(add);
 add++;
 count--;
 ptr++;

 }
 break;
```

实际上，这不是完全正确. 内存范围是很小和很频繁的使用，以至于内核在启动时建立页表来存取这些地址. 但是，这个用来存取它们的虚拟地址不是同一个物理地址，并且因此无论如何需要 ioremap.

下 2 个设备是 /dev/sillyw (次编号 1) 和 /dev/silly1 (次编号 2). 它们表现象 /dev/sillyb, 除了它们使用 16-位 和 32-位 函数. 这是 silly1 的写实现, 又一次部分 switch:

```
case M_32:
 while (count >= 4) {
 iowrite8(*(u32 *)ptr, add);
 add += 4;
 count -= 4;
 ptr += 4;

 }
 break;
```

最后的设备是 /dev/sillycp (次编号 3), 它使用 memcpy_*io 函数来进行同样的任务. 这是它的读实现的核心:

```
case M_memcpy:
 memcpy_fromio(ptr, add, count);
 break;
```

因为 ioremap 用来提供对 ISA 内存区的存取, silly 必须调用 iounmap 当模块卸载时:

```
iounmap(io_base);
```

9.4.6. isa_readb 和其友

看一下内核源码会展现另一套函数, 有如 isa_readb 的名子. 实际上, 每个刚才描述的函数都有一个 isa_ 对等体. 这些函数提供对 ISA 内存的存取不需要一个单独的 ioremap 步骤. 但是, 来自内核开发者的话, 是这些函数打算用来作为暂时的驱动移植辅助, 并且它可能将来消失. 因此, 你应当避免使用它们.

9.5. 快速参考

本章介绍下列与硬件管理相关的符号:

```
#include <linux/kernel.h>
void barrier(void)
```

这个"软件"内存屏蔽要求编译器对待所有内存是跨这个指令而非易失的.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

硬件内存屏障. 它们请求 CPU(和编译器)来检查所有的跨这个指令的内存读，写，或都有.

```
#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
```

用来读和写 I/O 端口的函数. 它们还可以被用户空间程序调用，如果它们有正当的权限来存取端口.

```
unsigned inb_p(unsigned port);
```

如果在一次 I/O 操作后需要一个小延时，你可以使用在前一项中介绍的这些函数的 6 个暂停对应部分；这些暂停函数有以 _p 结尾的名子.

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

这些"字串函数"被优化为传送数据从一个输入端口到一个内存区，或者其他的方式. 这些传送通过读或写到同一端口 count 次来完成.

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
int check_region(unsigned long start, unsigned long len);
```

I/O 端口的资源分配器. 这个检查函数成功返回 0 并且在错误时小于 0.

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
int check_mem_region(unsigned long start, unsigned long len);
```

为内存区处理资源分配的函数

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr);
```

ioremap 重映射一个物理地址范围到处理器的虚拟地址空间，使它对内核可用. iounmap 释放映射当不再需要它时.

```
#include <asm/io.h>
```

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

用来使用 I/O 内存的存取者函数.

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

I/O 内存原语的"重复"版本.

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);
```

旧的, 类型不安全的存取 I/O 内存的函数.

```
void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);
```

一个想对待 I/O 端口如同它们是 I/O 内存的驱动作者, 可以传递它们的端口给 ioport_map. 这个映射应当在不需要的时候恢复(使用 ioport_unmap)

第 10 章 中断处理

尽管一些设备可只使用它们的 I/O 区来控制，大部分真实的设备比那个要复杂点. 设备不得不和外部世界打交道，常常包括诸如旋转的磁盘，移动的磁带，连到远处的线缆，等等. 很多必须在一个时间片中完成，不同于，并且远慢于处理器. 因为几乎一直是不希望使处理器等待外部事件，对于设备必须有一种方法使处理器知道有事情发生了.

当然，那种方法是中断. 一个中断不过是一个硬件在它需要处理器的注意时能够发出的信号. Linux 处理中断非常类似它处理用户空间信号的方式. 对大部分来说，一个驱动只需要为它的设备中断注册一个处理函数，并且当它们到来时正确处理它们. 当然，在这个简单图像之下有一些复杂；特别地，中断处理有些受限于它们能够进行的动作，这是它们如何运行而导致的结果.

没有一个真实的硬件设备来产生中断，就难演示中断的使用. 因此，本章使用的例子代码使用并口工作. 这些端口在现代硬件上开始变得稀少，但是，运气地，大部分人仍然能够有一个有可用的端口的系统. 我们将使用来自上一章的简短模块；添加一小部分它能够产生并处理来自并口的中断. 模块的名子，short，实际上意味着 short int (它是 C，对不?)，来提醒我们它处理中断.

但是，在我们进入主题之前，是时候提出一个注意事项. 中断处理，由于它们的特性，与其他的代码并行地运行. 因此，它们不可避免地引起并发问题和对数据结构和硬件的竞争. 如果你屈服于诱惑以越过第 5 章的讨论，我们理解. 但是我们也建议你转回去并且现在看一下. 一个坚实的并发控制技术的理解是重要的，在使用中断时.

10.1. 准备并口

尽管并口简单，它能够触发中断. 这个能力被打印机用来通知 lp 驱动它准备好接收缓存中的下一个字符.

如同大部分设备，并口实际上不产生中断，在它被指示这样作之前；并口标准规定设置 port 2 (0x37a, 0x27a, 或者任何)的 bit 4 就使能中断报告. short 在模块初始化时进行一个简单的 outb 调用来设置这个位.

一旦中断使能，任何时候在管脚 10 (所谓的 ACK 位)上的电信号从低变到高，并口产生一个中断. 最简单的方法来强制接口产生中断(没有挂一个打印机到端口)是连接并口连接器的管脚 9 和 管脚 10. 一根短线，插到你的系统后面的并口连接器的合适的孔中，就建立这个连接. 并口外面的管脚图示于图并口的管脚.

管脚 9 是并口数据字节的最高位. 如果你写二进制数据到 /dev/short0，你产生几个中断. 然而，写 ASCII 文本到这个端口不会产生任何中断，因为 ASCII 字符集没有最高位置位的项.

如果你宁愿避免连接管脚到一起，而你手上确实有一台打印机，你可用使用一个真正的打印机来运行例子中断处理，如同下面展示的. 但是，注意我们介绍的探测函数依赖管脚 9 和管脚 10 之间的跳线在位置上，并且你需要它使用你的代码来试验探测.

10.2. 安装一个中断处理

如果你想实际地"看到"产生的中断，向硬件设备写不足够；一个软件处理必须在系统中配置．如果 Linux 内核还没有被告知来期待你的中断，它简单地确认并忽略它．

中断线是一个宝贵且常常有限的资源，特别当它们只有 15 或者 16 个时．内核保持了中断线的一个注册，类似于 I/O 端口的注册．一个模块被希望来请求一个中断通道(或者 IRQ，对于中断请求)，在使用它之前，并且当结束时释放它．在很多情况下，也希望模块能够与其他驱动共享中断线，如同我们将看到的．下面的函数，声明在 <linux/interrupt.h>，实现中断注册接口：

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,

                const char *dev_name,
                void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

从 request_irq 返回给请求函数的返回值或者是 0 指示成功，或者是一个负的错误码，如同平常．函数返回 -EBUSY 来指示另一个驱动已经使用请求的中断线是不寻常的．函数的参数如下：

unsigned int irq

请求的中断号

irqreturn_t (*handler)

安装的处理函数指针．我们在本章后面讨论给这个函数的参数以及它的返回值．

unsigned long flags

如你会希望的，一个与中断管理相关的选项的位掩码(后面描述)．

const char *dev_name

这个传递给 request_irq 的字串用在 /proc/interrupts 来显示中断的拥有者(下一节看到)

void *dev_id

用作共享中断线的指针．它是一个独特的标识，用在当释放中断线时以及可能还被驱动用来指向它自己的私有数据区(来标识哪个设备在中断)．如果中断没有被共享，dev_id 可以设置为 NULL，但是使用这个项指向设备结构不管如何是个好主意．我们将在"实现一个处理"一节中看到 dev_id 的一个实际应用．

flags 中可以设置的位如下：

SA_INTERRUPT

当置位了, 这表示一个"快速"中断处理. 快速处理在当前处理器上禁止中断来执行(这个主题在"快速和慢速处理"一节涉及).

SA_SHIRQ

这个位表示中断可以在设备间共享. 共享的概念在"中断共享"一节中略述.

SA_SAMPLE_RANDOM

这个位表示产生的中断能够有贡献给 /dev/random 和 /dev/urandom 使用的加密池. 这些设备在读取时返回真正的随机数并且设计来帮助应用程序软件为加密选择安全钥. 这样的随机数从一个由各种随机事件贡献的加密池中提取的. 如果你的设备以真正随机的时间产生中断, 你应当设置这个标志. 如果, 另一方面, 你的中断是可预测的(例如, 一个帧抓取器的场消隐), 这个标志不值得设置 -- 它无论如何不会对系统加密有贡献. 可能被攻击者影响的设备不应当设置这个标志; 例如, 网络驱动易遭受从外部计时的可预测报文并且不应当对加密池有贡献. 更多信息看 drivers/char/random.c 的注释.

中断处理可以在驱动初始化时安装或者在设备第一次打开时. 尽管从模块的初始化函数中安装中断处理可能听来是个好主意, 它常常不是, 特别当你的设备不共享中断. 因为中断线数目是有限的, 你不想浪费它们. 你可以轻易使你的系统中设备数多于中断数.如果一个模块在初始化时请求一个 IRQ, 它阻止了任何其他的驱动使用这个中断, 甚至这个持有它的设备从不被使用. 在设备打开时请求中断, 另一方面, 允许某些共享资源.

例如, 可能与一个 modem 在同一个中断上运行一个帧抓取器, 只要你不同时使用这 2 个设备. 对用户来说是很普通的在系统启动时为一个特殊设备加载模块, 甚至这个设备很少用到. 一个数据获取技巧可能使用同一个中断作为第 2 个串口. 虽然不是太难避免在数据获取时联入你的互联网服务提供商(ISP), 被迫卸载一个模块为了使用 modem 确实令人不快.

调用 request_irq 的正确位置是当设备第一次打开时, 在硬件被指示来产生中断前. 调用 free_irq 的位置是设备最后一次被关闭时, 在硬件被告知不要再中断处理器之后. 这个技术的缺点是你需要保持一个每设备的打开计数, 以便于你知道什么时候中断可以被禁止.

尽管这个讨论, short 还在加载时请求它的中断线. 这样做是为了你可以运行测试程序而不必运行一个额外的进程来保持设备打开. short, 因此, 从它的初始化函数(short_init)请求中断, 不是在 short_open 中做, 象一个真实设备驱动.

下面代码请求的中断是 short_irq. 变量的真正赋值(即, 决定使用哪个 IRQ)在后面显示, 因为它和现在的讨论无关. short_base 是使用的并口 I/O 基地址; 接口的寄存器 2 被写入来使能中断报告.

```
if (short_irq >= 0)
{
```

```
        result = request_irq(short_irq, short_interrupt,
                             SA_INTERRUPT, "short", NULL);
        if (result) {
                printk(KERN_INFO "short: can't get assigned irq %i\n",
                       short_irq);

                short_irq = -1;
        } else { /* actually enable it -- assume this *is* a parallel port */
                outb(0x10,short_base+2);
        }
}
```

代码显示, 安装的处理是一个快速处理(SA_INTERRUPT), 不支持中断共享(SA_SHIRQ 没有), 并且不对系统加密有贡献(SA_SAMPLE_RANDOM 也没有). outb 调用接着为并口使能中断报告.

由于某些合理原因, i386 和 x86_64 体系定义了一个函数来询问一个中断线的能力:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

这个函数当试图分配一个给定中断成功时返回一个非零值. 但是, 注意, 在 can_request_irq 和 request_irq 的调用之间事情可能一直改变.

10.2.1. /proc 接口

无论何时一个硬件中断到达处理器, 一个内部的计数器递增, 提供了一个方法来检查设备是否如希望地工作. 报告的中断显示在 /proc/interrupts. 下面的快照取自一个双处理器 Pentium 系统:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
        CPU0     CPU1
 0:  4848108       34    IO-APIC-edge  timer
 2:        0        0          XT-PIC  cascade
 8:        3        1    IO-APIC-edge  rtc
 10:    4335        1   IO-APIC-level  aic7xxx
 11:    8903        0   IO-APIC-level  uhci_hcd
 12:      49        1    IO-APIC-edge  i8042
NMI:       0        0
LOC: 4848187  4848186
ERR:       0
MIS:       0
```

第一列是 IRQ 号. 你能够从没有的 IRQ 中看到这个文件只显示对应已安装处理的中断. 例如, 第一个串口(使用中断号 4)没有显示, 指示 modem 没在使用. 事实上, 即便如果 modem 已更早使用了, 但是在这个快照时间没有使用, 它不会显示在这个文件中; 串口表现很好并且在设备关闭时释放它们的中断处理.

/proc/interrupts 的显示展示了有多少中断硬件递交给系统中的每个 CPU. 如同你可从输出看到的, Linux 内核常常在第一个 CPU 上处理中断, 作为一个使 cache 局部性最大化的方法.[37] 最后 2 列给出关于处理中断的可编程中断控制器的信息(驱动编写者不必关心), 以及已注册的中断处理的设备的名子(如同在给 request_irq 的参数 dev_name 中指定的).

/proc 树包含另一个中断有关的文件, /proc/stat; 有时你会发现一个文件更加有用并且有时你会喜欢另一个. /proc/stat 记录了几个关于系统活动的低级统计量, 包括(但是不限于)自系统启动以来收到的中断数. stat 的每一行以一个文本字串开始, 是该行的关键词; intr 标志是我们在找的. 下列(截短了)快照是在前一个后马上取得的:

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

第一个数是所有中断的总数, 而其他每一个代表一个单个 IRQ 线, 从中断 0 开始. 所有的计数跨系统中所有处理器而汇总的. 这个快照显示, 中断号 4 已使用 4907 次, 尽管当前没有安装处理. 如果你在测试的驱动请求并释放中断在每个打开和关闭循环, 你可能发现 /proc/stat 比 /proc/interrupts 更加有用.

2 个文件的另一个不同是, 中断不是体系依赖的(也许, 除了末尾几行), 而 stat 是; 字段数依赖内核之下的硬件. 可用的中断数目少到在 SPARC 上的 15 个, 多到 IA-64 上的 256 个, 并且其他几个系统都不同. 有趣的是要注意, 定义在 x86 中的中断数当前是 224, 不是你可能期望的 16; 如同在 include/asm-i386/irq.h 中解释的, 这依赖 Linux 使用体系的限制, 而不是一个特定实现的限制(例如老式 PC 中断控制器的 16 个中断源).

下面是一个 /proc/interrupts 的快照, 取自一台 IA-64 系统. 如你所见, 除了不同硬件的通用中断源的路由, 输出非常类似于前面展示的 32-位 系统的输出.

```
          CPU0       CPU1
 27:      1705      34141   IO-SAPIC-level   qla1280
 40:         0          0   SAPIC                                   perfmon
 43:       913       6960   IO-SAPIC-level   eth0
 47:     26722        146   IO-SAPIC-level   usb-uhci
 64:         3          6   IO-SAPIC-edge    ide0
 80:         4          2   IO-SAPIC-edge    keyboard
 89:         0          0   IO-SAPIC-edge    PS/2 Mouse
239:   5606341   5606052            SAPIC    timer

254:   67575   52815   SAPIC   IPI
NMI:   0   0
ERR:   0
```

10.2.2. 自动检测 IRQ 号

驱动在初始化时最有挑战性的问题中的一个是如何决定设备要使用哪个 IRQ 线. 驱动需要信息来正确安装处理. 尽管程序员可用请求用户在加载时指定中断号, 这是个坏做法, 因为大部分时间用户不知道这个号, 要么因为他不配置跳线要么因为设备是无跳线的. 大

部分用户希望他们的硬件"仅仅工作"并且不感兴趣如中断号的问题. 因此自动检测中断号是一个驱动可用性的基本需求.

有时自动探测依赖知道一些设备有很少改变的缺省动作的特性. 在这个情况下, 驱动可能假设缺省值适用. 这确切地就是 short 如何缺省对并口动作的. 实现是直接的, 如 short 自身显示的:

```
if (short_irq < 0) /* not yet specified: force the default on */
 switch(short_base) {
 case 0x378: short_irq = 7; break;
 case 0x278: short_irq = 2; break;
 case 0x3bc: short_irq = 5; break;
 }
```

代码根据选择的 I/O 基地址赋值中断号, 而允许用户在加载时覆盖缺省值, 使用如:

```
insmod ./short.ko irq=x
short_base defaults to 0x378, so short_irq defaults to 7.
```

有些设备设计得更高级并且简单地"宣布"它们要使用的中断. 在这个情况下, 驱动获取中断号通过从设备的一个 I/O 端口或者 PCI 配置空间读一个状态字节. 当目标设备是一个有能力告知驱动它要使用哪个中断的设备时, 自动探测中断号只是意味着探测设备, 探测中断没有其他工作要做. 幸运的是大部分现代硬件这样工作; 例如, PCI 标准解决了这个问题通过要求外设来声明它们要使用哪个中断线. PCI 标准在 12 章讨论.

不幸的是, 不是每个设备是对程序员友好的, 并且自动探测可能需要一些探测. 这个技术非常简单: 驱动告知设备产生中断并且观察发生了什么. 如果所有事情进展地好, 只有一个中断线被激活.

尽管探测在理论上简单的, 实际的实现可能不清晰. 我们看 2 种方法来进行这个任务: 调用内核定义的帮助函数和实现我们自己的版本.

10.2.2.1. 内核协助的探测

Linux 内核提供了一个低级设施来探测中断号. 它只为非共享中断, 但是大部分能够在共享中断状态工作的硬件提供了更好的方法来尽量发现配置的中断号.这个设施包括 2 个函数, 在<linux/interrupt.h> 中声明(也描述了探测机制).

```
unsigned long probe_irq_on(void);
```

> 这个函数返回一个未安排的中断的位掩码. 驱动必须保留返回的位掩码, 并且在后面传递给 probe_irq_off. 在这个调用之后, 驱动应当安排它的设备产生至少一次中断.

```
int probe_irq_off(unsigned long);
```

在设备已请求一个中断后，驱动调用这个函数，作为参数传递之前由 probe_irq_on 返回的位掩码. probe_irq_off 返回在"probe_on"之后发出的中断号. 如果没有中断发生，返回 0 (因此，IRQ 0 不能探测，但是没有用户设备能够在任何支持的体系上使用它). 如果多于一个中断发生(模糊的探测)，probe_irq_off 返回一个负值.

程序员应当小心使能设备上的中断，在调用 probe_irq_on 之后以及在调用 probe_irq_off 后禁止它们. 另外，你必须记住服务你的设备中挂起的中断，在 probe_irq_off 之后.

short 模块演示了如何使用这样的探测. 如果你加载模块使用 probe=1，下列代码被执行来探测你的中断线，如果并口连接器的管脚 9 和 10 连接在一起:

```
int count = 0;
do
{
        unsigned long mask;
        mask = probe_irq_on();
        outb_p(0x10,short_base+2); /* enable reporting */
        outb_p(0x00,short_base); /* clear the bit */
        outb_p(0xFF,short_base); /* set the bit: interrupt! */
        outb_p(0x00,short_base+2); /* disable reporting */
        udelay(5); /* give it some time */
        short_irq = probe_irq_off(mask);

        if (short_irq == 0) { /* none of them? */
                printk(KERN_INFO "short: no irq reported by probe\n");
                short_irq = -1;
        }

        /*
         * if more than one line has been activated, the result is
         * negative. We should service the interrupt (no need for lpt port)
         * and loop over again. Loop at most five times, then give up
         */
} while (short_irq < 0 && count++ < 5);
if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

注意 udelay 的使用，在调用 probe_irq_off 之前. 依赖你的处理器的速度，你可能不得不等待一小段时间来给中断时间来真正被递交.

探测可能是一个长时间的任务. 虽然对于 short 这不是真的，例如，探测一个帧抓取器，需要一个至少 20 ms 的延时(对处理器是一个时代)，并且其他的设备可能要更长. 因此，最好只探测中断线一次，在模块初始化时，独立于你是否在设备打开时安装处理(如同你应当做的)，或者在初始化函数当中(这个不推荐).

有趣的是注意在一些平台上(PoweerPC, M68K, 大部分 MIPS 实现, 以及 2 个 SPARC 版本)探测是不必要的, 并且, 因此, 之前的函数只是空的占位者, 有时称为"无用的 ISA 废话". 在其他平台上, 探测只为 ISA 设备实现. 无论如何, 大部分体系定义了函数(即便它们是空的)来简化移植现存的设备驱动.

10.2.2.2. Do-it-yourself 探测

探测也可以在驱动自身实现没有太大麻烦. 它是一个少有的驱动必须实现它自己的探测, 但是看它是如何工作的能够给出对这个过程的内部认识. 为此目的, short 模块进行 do-it-yourself 的 IRQ 线探测, 如果它使用 probe=2 加载.

这个机制与前面描述的相同: 使能所有未使用的中断, 接着等待并观察发生什么. 我们能够, 然而, 利用我们对设备的知识. 常常地一个设备能够配置为使用一个 IRQ 号从 3 个或者 4 个一套; 只探测这些 IRQ 使我们能够探测正确的一个, 不必测试所有的可能中断.

short 实现假定 3, 5, 7, 和 9 是唯一可能的 IRQ 值. 这些数实际上是一些并口设备允许你选择的数.

下面的代码通过测试所有"可能的"中断并且查看发生的事情来探测中断. trials 数组列出要尝试的中断, 以 0 作为结尾标志; tried 数组用来跟踪哪个处理实际上被这个驱动注册.

```
int trials[] =
        {
                3, 5, 7, 9, 0
        };
int tried[]  = {0, 0, 0, 0, 0};
int i, count = 0;

/*
 * install the probing handler for all possible lines. Remember
 * the result (0 for success, or -EBUSY) in order to only free
 * what has been acquired */
for (i = 0; trials[i]; i++)
        tried[i] = request_irq(trials[i], short_probing,
                               SA_INTERRUPT, "short probe", NULL);

do
{
        short_irq = 0; /* none got, yet */
        outb_p(0x10,short_base+2); /* enable */
        outb_p(0x00,short_base);
        outb_p(0xFF,short_base); /* toggle the bit */
        outb_p(0x00,short_base+2); /* disable */
        udelay(5); /* give it some time */

        /* the value has been set by the handler */
        if (short_irq == 0) { /* none of them? */

                printk(KERN_INFO "short: no irq reported by probe\n");
```

```
        }
        /*
        * If more than one line has been activated, the result is
        * negative. We should service the interrupt (but the lpt port
        * doesn't need it) and loop over again. Do it at most 5 times
        */
} while (short_irq <=0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i = 0; trials[i]; i++)
        if (tried[i] == 0)
                free_irq(trials[i], NULL);

if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

你可能事先不知道"可能的" IRQ 值是什么. 在这个情况, 你需要探测所有空闲的中断, 不是限制你自己在几个 trials[]. 为探测所有的中断, 你不得不从 IRQ 0 到 IRQ NR_IRQS-1 探测, 这里 NR_IRQS 在 <asm/irq.h> 中定义并且是独立于平台的.

现在我们只缺少探测处理自己了. 处理者的角色是更新 short_irq, 根据实际收到哪个中断. short_irq 中的 0 值意味着"什么没有", 而一个负值意味着"模糊的". 这些值选择来和 probe_irq_off 相一致并且允许同样的代码来调用任一种 short.c 中的探测.

```
irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
{

    if (short_irq == 0) short_irq = irq;   /* found */
 if (short_irq != irq) short_irq = -irq; /* ambiguous */
 return IRQ_HANDLED;
}
```

处理的参数在后面描述. 知道 irq 是在处理的中断应当是足够的来理解刚刚展示的函数.

10.2.3. 快速和慢速处理

老版本的 Linux 内核尽了很大努力来区分"快速"和"慢速"中断. 快速中断是那些能够很快处理的, 而处理慢速中断要特别地长一些. 慢速中断可能十分苛求处理器, 并且它值得在处理的时候重新使能中断. 否则, 需要快速注意的任务可能被延时太长.

在现代内核中, 快速和慢速中断的大部分不同已经消失. 剩下的仅仅是一个: 快速中断(那些使用 SA_INTERRUPT 被请求的)执行时禁止所有在当前处理器上的其他中断. 注意其他的处理器仍然能够处理中断, 尽管你从不会看到 2 个处理器同时处理同一个 IRQ.

这样, 你的驱动应当使用哪个类型的中断? 在现代系统上, SA_INTERRUPT 只是打算用在几个, 特殊的情况例如时钟中断. 除非你有一个充足的理由来运行你的中断处理在禁止其他中断情况下, 你不应当使用 SA_INTERRUPT.

这个描述应当满足大部分读者, 尽管有人喜好硬件并且对她的计算机有经验可能有兴趣深入一些. 如果你不关心内部的细节, 你可跳到下一节.

10.2.3.1. x86 上中断处理的内幕

这个描述是从 arch/i386/kernel/irq.c, arch/i386/kernel/ apic.c, arch/i386/kernel/entry.S, arch/i386/kernel/i8259.c, 和 include/asm-i386/hw_irq.h 它们出现于 2.6 内核而推知的; 尽管一般的概念保持一致, 硬件细节在其他平台上不同.

中断处理的最低级是在 entry.S, 一个汇编语言文件处理很多机器级别的工作. 通过一点汇编器的技巧和一些宏定义, 一点代码被安排到每个可能的中断. 在每个情况下, 这个代码将中断号压栈并且跳转到一个通用段, 称为 do_IRQ, 在 irq.c 中定义.

do_IRQ 做的第一件事是确认中断以便中断控制器能够继续其他事情. 它接着获取给定 IRQ 号的一个自旋锁, 因此阻止任何其他 CPU 处理这个 IRQ. 它清除几个状态位(包括称为 IRQ_WAITING 的一个, 我们很快会看到它)并且接着查看这个特殊 IRQ 的处理者. 如果没有处理者, 什么不作; 自旋锁释放, 任何挂起的软件中断被处理, 最后 do_IRQ 返回.

常常, 但是, 如果一个设备在中断, 至少也有一个处理者注册给它的 IRQ. 函数 handle_IRQ_event 被调用来实际调用处理者. 如果处理者是慢速的(SA_INTERRUPT 没有设置), 中断在硬件中被重新使能, 并且调用处理者. 接着仅仅是清理, 运行软件中断, 以及回到正常的工作. "常规工作"很可能已经由于中断而改变了(处理者可能唤醒一个进程, 例如), 因此从中断中返回的最后的事情是一个处理器的可能的重新调度.

探测 IRQ 通过设置 IRQ_WAITING 状态位给每个当前缺乏处理者的 IRQ 来完成. 当中断发生, do_IRQ 清除这个位并且接着返回, 因为没有注册处理者. probe_irq_off, 当被一个函数调用, 需要只搜索不再有 IRQ_WAITING 设置的 IRQ.

10.2.4. 实现一个处理

至今, 我们已学习了注册一个中断处理, 但是没有编写一个. 实际上, 对于一个处理者, 没什么不寻常的 -- 它是普通的 C 代码.

唯一的特别之处是一个处理者在中断时运行, 因此, 它能做的事情遭受一些限制. 这些限制与我们在内核定时器上看到的相同. 一个处理者不能传递数据到或者从用户空间, 因为它不在进程上下文执行. 处理者也不能做任何可能睡眠的事情, 例如调用 wait_event, 使用除 GFP_ATOMIC 之外任何东西来分配内存, 或者加锁一个旗标. 最后, 处理者不能调用调度.

一个中断处理的角色是给它的设备关于中断接收的回应并且读或写数据, 根据被服务的中断的含义. 第一步常常包括清除接口板上的一位; 大部分硬件设备不产生别的中断直到它们的"中断挂起"位被清除. 根据你的硬件如何工作的, 这一步可能需要在最后做而不是开始; 这里没有通吃的规则. 一些设备不需要这步, 因为它们没有一个"中断挂起"位; 这样的设备是一少数, 尽管并口是其中之一. 由于这个理由, short 不必清除这样一个位.

一个中断处理的典型任务是唤醒睡眠在设备上的进程, 如果中断指示它们在等待的事件, 例如新数据的到达.

为坚持帧抓取者的例子，一个进程可能请求一个图像序列通过连续读设备；读调用阻塞在读取每个帧之前，而中断处理唤醒进程一旦每个新帧到达. 这个假定抓取器中断处理器来指示每个新帧的成功到达.

程序员应当小心编写一个函数在最小量的时间内执行，不管是一个快速或慢速处理者. 如果需要进行长时间计算，最好的方法是使用一个 tasklet 或者 workqueue 来调度计算在一个更安全的时间(我们将在"上和下半部"一节中见到工作如何被延迟.).

我们在 short 中的例子代码响应中断通过调用 do_gettimeofday 和 打印当前时间到一个页大小的环形缓存. 它接着唤醒任何读进程，因为现在有数据可用来读取.

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
        {
                struct timeval tv;
                int written;
                do_gettimeofday(&tv);
                /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
                written = sprintf((char *)short_head,"%08u.%06u\n",
                                  (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
                BUG_ON(written != 16);
                short_incr_bp(&short_head, written);
                wake_up_interruptible(&short_queue); /* awake any reading process */
                return IRQ_HANDLED;
        }
```

这个代码，尽管简单，代表了一个中断处理的典型工作. 依次地，它称为 short_incr_bp，定义如下:

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
        unsigned long new = *index + delta;
        barrier(); /* Don't optimize these two together */
        *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

这个函数已经仔细编写来回卷指向环形缓存的指针，没有暴露一个不正确的值. 这里的 barrier 调用来阻止编译器在这个函数的其他 2 行之间优化. 如果没有 barrier，编译器可能决定优化掉 new 变量并且直接赋值给 *index. 这个优化可能暴露一个 index 的不正确值一段时间，在它回卷的地方. 通过小心阻止对其他线程可见的不一致的值，我们能够安全操作环形缓存指针而不用锁.

用来读取中断时填充的缓存的设备文件是 /dev/shortint. 这个设备特殊文件，同 /dev/shortprint 一起，不在第 9 章介绍，因为它的使用对中断处理是特殊的. /dev/shortint 内部特别地为中断产生和报告剪裁过. 写到设备会每隔一个字节产生一个中断；读取设备给出了每个中断被报告的时间.

如果你连接并口连接器的管脚 9 和 10，你可产生中断通过拉高并口数据字节的高位. 这可通过写二进制数据到 /dev/short0 或者通过写任何东西到 /dev/shortint 来完成.

[38]下列代码为 /dev/shortint 实现读和写:

```
ssize_t short_i_read (struct file *filp, char __user *buf, size_t count,
                      loff_t *f_pos)
{
        int count0;
        DEFINE_WAIT(wait);

        while (short_head == short_tail)
        {
                prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
                if (short_head == short_tail)
                        schedule();
                finish_wait(&short_queue, &wait);
                if (signal_pending (current)) /* a signal arrived */
                        return -ERESTARTSYS; /* tell the fs layer to handle it */
        } /* count0 is the number of readable data bytes */ count0 = short_head -
short_tail;
        if (count0 < 0) /* wrapped */
                count0 = short_buffer + PAGE_SIZE - short_tail;
        if (count0 < count)
                count = count0;

        if (copy_to_user(buf, (char *)short_tail, count))
                return -EFAULT;
        short_incr_bp (&short_tail, count);
        return count;
}
ssize_t short_i_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
        int written = 0, odd = *f_pos & 1;
        unsigned long port = short_base; /* output to the parallel data latch */
        void *address = (void *) short_base;

        if (use_mem)
        {
                while (written < count)
                        iowrite8(0xff * ((++written + odd) & 1), address);
        }
         else
        {
                while (written < count)
                        outb(0xff * ((++written + odd) & 1), port);
        }

        *f_pos += count;
        return written;
}
```

其他设备特殊文件，/dev/shortprint，使用并口来驱动一个打印机；你可用使用它，如果你想避免连接一个 D-25 连接器管脚 9 和 10. shortprint 的写实现使用一个环形缓存来存储要打印的数据，而写实现是刚刚展示的那个(因此你能够读取你的打印机吃进每个字符用的时间).

为了支持打印机操作，中断处理从刚刚展示的那个已经稍微修改，增加了发送下一个数据字节到打印机的能力，如果没有更多数据传送.

10.2.5. 处理者的参数和返回值

尽管 short 忽略了它们，一个传递给一个中断处理的参数: irq，dev_id，和 regs. 我们看一下每个的角色.

中断号(int irq)作为你可能在你的 log 消息中打印的信息是有用的，如果有. 第二个参数，void *dev_id，是一类客户数据; 一个 void* 参数传递给 request_irq，并且同样的指针接着作为一个参数传回给处理者，当中断发生时. 你常常传递一个指向你的在 dev_id 中的设备数据结构的指针，因此一个管理相同设备的几个实例的驱动不需要任何额外的代码，在中断处理中找出哪个设备要负责当前的中断事件.

这个参数在中断处理中的典型使用如下:

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        struct sample_dev *dev = dev_id;

        /* now `dev' points to the right hardware item */
        /* .... */
}
```

和这个处理者关联的典型的打开代码看来如此:

```
static void sample_open(struct inode *inode, struct file *filp)
{
        struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
        request_irq(dev->irq, sample_interrupt,

                        0 /* flags */, "sample", dev /* dev_id */);
        /*....*/
        return 0;

}
```

最后一个参数，struct pt_regs *regs，很少用到. 它持有一个处理器的上下文在进入中断状态前的快照. 寄存器可用来监视和调试; 对于常规地设备驱动任务，正常地不需要它们.

中断处理应当返回一个值指示是否真正有一个中断要处理. 如果处理者发现它的设备确实需要注意，它应当返回 IRQ_HANDLED; 否则返回值应当是 IRQ_NONE. 你也可产生返回值，使用这个宏:

```
IRQ_RETVAL(handled)
```

这里，handled 是非零，如果你能够处理中断. 内核用返回值来检测和抑制假中断. 如果你的设备没有给你方法来告知是否它确实中断，你应当返回 IRQ_HANDLED.

10.2.6. 使能和禁止中断

有时设备驱动必须阻塞中断的递交一段时间(希望地短)(我们在第 5 章的 "自旋锁"一节看到过这样的一个情况). 常常, 中断必须被阻塞当持有一个自旋锁来避免死锁系统时. 有几个方法来禁止不涉及自旋锁的中断. 但是在我们讨论它们之前, 注意禁止中断应当是一个相对少见的行为, 即便在设备驱动中, 并且这个技术应当从不在驱动中用做互斥机制.

10.2.6.1. 禁止单个中断

有时(但是很少!)一个驱动需要禁止一个特定中断线的中断递交. 内核提供了 3 个函数为此目的, 所有都声明在 <asm/irq.h>. 这些函数是内核 API 的一部分, 因此我们描述它们, 但是它们的使用在大部分驱动中不鼓励. 在其他的中, 你不能禁止共享的中断线, 并且, 在现代的系统中, 共享的中断是规范. 已说过的, 它们在这里:

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

调用任一函数可能更新在可编程控制器(PIC)中的特定 irq 的掩码, 因此禁止或使能跨所有处理器的特定 IRQ. 对这些函数的调用能够嵌套 -- 如果 disable_irq 被连续调用 2 次, 需要 2 个 enable_irq 调用在 IRQ 被真正重新使能前. 可能调用这些函数从一个中断处理中, 但是在处理它时使能你自己的 IRQ 常常不是一个好做法.

disable_irq 不仅禁止给定的中断, 还等待一个当前执行的中断处理结束, 如果有. 要知道如果调用 disable_irq 的线程持有中断处理需要的任何资源(例如自旋锁), 系统可能死锁. disable_irq_nosync 与 disable_irq 不同, 它立刻返回. 因此, 使用 disable_irq_nosync 快一点, 但是可能使你的设备有竞争情况.

但是为什么禁止中断? 坚持说并口, 我们看一下 plip 网络接口. 一个 plip 设备使用裸并口来传送数据. 因为只有 5 位可以从并口连接器读出, 它们被解释为 4 个数据位和一个时钟/握手信号. 当一个报文的第一个 4 位被 initiator (发送报文的接口) 传送, 时钟线被拉高, 使接收接口来中断处理器. plip 处理者接着被调用来处理新到达的数据.

在设备已经被提醒了后, 数据传送继续, 使用握手线来传送数据到接收接口(这可能不是最好的实现, 但是有必要与使用并口的其他报文驱动兼容). 如果接收接口不得不为每个接收的字节处理 2 次中断, 性能可能不可忍受. 因此, 驱动在接收报文的时候禁止中断; 相反, 一个查询并延时的循环用来引入数据.

类似地, 因为从接收器到发送器的握手线用来确认数据接收, 发送接口禁止它的 IRQ 线在报文发送时.

10.2.6.2. 禁止所有中断

如果你需要禁止所有中断如何? 在 2.6 内核, 可能关闭在当前处理器上所有中断处理, 使用任一个下面 2 个函数(定义在 <asm/system.h>):

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

一个对 local_irq_save 的调用在当前处理器上禁止中断递交, 在保存当前中断状态到 flags 之后. 注意, flags 是直接传递, 不是通过指针. local_irq_disable 关闭本地中断递交而不保存状态; 你应当使用这个版本只在你知道中断没有在别处被禁止.

完成打开中断, 使用:

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

第一个版本恢复由 local_irq_save 存储于 flags 的状态, 而 local_irq_enable 无条件打开中断. 不象 disable_irq, local_irq_disable 不跟踪多次调用. 如果调用链中有多于一个函数可能需要禁止中断, 应该使用 local_irq_save.

在 2.6 内核, 没有方法全局性地跨整个系统禁止所有的中断. 内核开发者决定, 关闭所有中断的开销太高, 并且在任何情况下没有必要有这个能力. 如果你在使用一个旧版本驱动, 它调用诸如 cli 和 sti, 你需要在它在 2.6 下工作前更新它为使用正确的加锁

---

[37] 尽管, 一些大系统明确使用中断平衡机制来在系统间分散中断负载.

[38] 这个 shortint 设备完成它的任务, 通过交替地写入 0x00 和 0xff 到并口.

10.3. 前和后半部

中断处理的一个主要问题是如何在处理中进行长时间的任务. 常常大量的工作必须响应一个设备中断来完成, 但是中断处理需要很快完成并且不使中断阻塞太长. 这 2 个需要(工作和速度)彼此冲突, 留给驱动编写者一点困扰.

Linux (许多其他系统一起)解决这个问题通过将中断处理分为 2 半. 所谓的前半部是实际响应中断的函数 -- 你使用 request_irq 注册的那个. 后半部是由前半部调度来延后执行的函数, 在一个更安全的时间. 最大的不同在前半部处理和后半部之间是所有的中断在后半部执行时都使能 -- 这就是为什么它在一个更安全时间运行. 在典型的场景中, 前半部保存设备数据到一个设备特定的缓存, 调度它的后半部, 并且退出: 这个操作非常快. 后半部接着进行任何其他需要的工作, 例如唤醒进程, 启动另一个 I/O 操作, 等等. 这种设置允许前半部来服务一个新中断而同时后半部仍然在工作.

几乎每个认真的中断处理都这样划分. 例如, 当一个网络接口报告有新报文到达, 处理者只是获取数据并且上推给协议层; 报文的实际处理在后半部进行.

Linux 内核有 2 个不同的机制可用来实现后半部处理, 我们都在第 7 章介绍. tasklet 常常是后半部处理的首选机制; 它们非常快, 但是所有的 tasklet 代码必须是原子的. tasklet 的可选项是工作队列, 它可能有一个更高的运行周期但是允许睡眠.

下面的讨论再次使用 short 驱动. 当使用一个模块选项加载时, short 能够被告知在前/后半部模式使用一个 tasklet 或者工作队列处理者来进行中断处理. 在这个情况下, 前半部快速地执行; 它简单地记住当前时间并且调度后半部处理. 后半部接着负责将时间编码并且唤醒任何可能在等待数据的用户进程.

10.3.1. Tasklet 实现

记住 tasklet 是一个特殊的函数, 可能被调度来运行, 在软中断上下文, 在一个系统决定的安全时间中. 它们可能被调度运行多次, 但是 tasklet 调度不累积; ; tasklet 只运行一次, 即便它在被投放前被重复请求. 没有 tasklet 会和它自己并行运行, 因为它只运行一次, 但是 tasklet 可以与 SMP 系统上的其他 tasklet 并行运行. 因此, 如果你的驱动有多个 tasklet, 它们必须采取某类加锁来避免彼此冲突.

tasklet 也保证作为函数运行在第一个调度它们的同一个 CPU 上. 因此, 一个中断处理可以确保一个 tasklet 在处理者结束前不会开始执行. 但是, 另一个中断当然可能在 tasklet 在运行时被递交, 因此, tasklet 和中断处理之间加锁可能仍然需要.

tasklet 必须使用 DECLARE_TASKLET 宏来声明:

```
DECLARE_TASKLET(name, function, data);
```

name 是给 tasklet 的名子, function 是调用来执行 tasklet (它带一个 unsigned long 参数并且返回 void)的函数, 以及 data 是一个 unsigned long 值来传递给 tasklet 函数.

short 驱动声明它的 tasklet 如下:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

函数 tasklet_schedule 用来调度一个 tasklet 运行. 如果 short 使用 tasklet=1 来加载, 它安装一个不同的中断处理来保存数据并且调度 tasklet 如下:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* cast to stop 'volatile' warning */
       short_incr_tv(&tv_head);
       tasklet_schedule(&short_tasklet);
       short_wq_count++; /* record that an interrupt arrived */
       return IRQ_HANDLED;
}
```

实际的 tasklet 函数, short_do_tasklet, 将在系统方便时很快执行. 如同前面提过, 这个函数进行处理中断的大量工作; 它看来如此:

```
void short_do_tasklet (unsigned long unused)
{
        int savecount = short_wq_count, written;
        short_wq_count = 0; /* we have already been removed from the queue */
```

```
        /*
        * The bottom half reads the tv array, filled by the top half,
        * and prints it to the circular text buffer, which is then consumed
        * by reading processes */
        /* First write the number of interrupts that occurred before this bh */
        written = sprintf((char *)short_head,"bh after %6i\n",savecount);
        short_incr_bp(&short_head, written);
        /*
        * Then, write the time values. Write exactly 16 bytes at a time,
        * so it aligns with PAGE_SIZE */

        do {
                written = sprintf((char *)short_head,"%08u.%06u\n",
                                  (int)(tv_tail->tv_sec % 100000000),
                                  (int)(tv_tail->tv_usec));
                short_incr_bp(&short_head, written);
                short_incr_tv(&tv_tail);
        } while (tv_tail != tv_head);

        wake_up_interruptible(&short_queue); /* awake any reading process */
}
```

在别的东西中，这个 tasklet 记录了从它上次被调用以来有多少中断到达. 一个如 short 一样的设备能够在短时间内产生大量中断，因此在后半部执行前有几个中断到达就不是不寻常的. 驱动必须一直准备这种可能性并且必须能够从前半部留下的信息中决定有多少工作要做.

10.3.2. 工作队列

回想，工作队列在将来某个时候调用一个函数，在一个特殊工作者进程的上下文中. 因为这个工作队列函数在进程上下文运行，它在需要时能够睡眠. 但是，你不能从一个工作队列拷贝数据到用户空间，除非你使用我们在 15 章演示的高级技术；工作者进程不存取任何其他进程的地址空间.

short 驱动，如果设置 wq 选项为一个非零值来加载，为它的后半部处理使用一个工作队列. 它使用系统缺省的工作队列，因此不要求特殊的设置代码；如果你的驱动有特别的运行周期要求(或者可能在工作队列函数长时间睡眠)，你可能需要创建你自己的，专用的工作队列. 我们确实需要一个 work_struct 结构，它声明和初始化使用下列:

```
static struct work_struct short_wq;
/* this line is in short_init() */
INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

我们的工作者函数是 short_do_tasklet，我们已经在前面一节看到.

当使用一个工作队列，short 还建立另一个中断处理，看来如此:

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        /* Grab the current time information. */
```

```
        do_gettimeofday((struct timeval *) tv_head);
        short_incr_tv(&tv_head);
        /* Queue the bh. Don't worry about multiple enqueueing */
        schedule_work(&short_wq);
        short_wq_count++; /* record that an interrupt arrived */
        return IRQ_HANDLED;
}
```

如你所见, 中断处理看来非常象这个 tasklet 版本, 除了它调用 schedule_work 来安排后半部处理.

10.4. 中断共享

中断冲突的概念几乎是 PC 体系的同义词. 过去, 在 PC 上的 IRQ 线不能服务多于一个设备, 并且它们从不足够. 结果, 失望的用户花费大量时间开着它们的计算机, 尽力找到一个方法来使它们所有的外设一起工作.

现代的硬件, 当然, 已经设计来允许中断共享; PCI 总线要求它. 因此, Linux 内核支持在所有总线上中断共享, 甚至是那些(例如 ISA 总线)传统上不被支持的. 2.6 内核的设备驱动应当编写来使用共享中断, 如果目标硬件能够支持这个操作模式. 幸运的是, 使用共享中断在大部分时间是容易的.

10.4.1. 安装一个共享的处理者

共享中断通过 request_irq 来安装就像不共享的一样, 但是有 2 个不同:

- SA_SHIRQ 位必须在 flags 参数中指定, 当请求中断时.
- dev_id 参数必须是独特的. 任何模块地址空间的指针都行, 但是 dev_id 明确地不能设置为 NULL.

内核保持着一个与中断相关联的共享处理者列表, 并且 dev_id 可认为是区别它们的签名. 如果 2 个驱动要在同一个中断上注册 NULL 作为它们的签名, 在卸载时事情可能就乱了, 在中断到的时候引发内核 oops. 由于这个理由, 如果在注册共享中断时传给了一个 NULL dev_id , 现代内核会大声抱怨. 当请求一个共享的中断, request_irq 成功, 如果下列之一是真:

- 中断线空闲.
- 所有这条线的已经注册的处理者也指定共享这个 IRQ.

无论何时 2 个或多个驱动在共享中断线, 并且硬件中断在这条线上中断处理器, 内核为这个中断调用每个注册的处理者, 传递它的 dev_id 给每个. 因此, 一个共享的处理者必须能够识别它自己的中断并且应当快速退出当它自己的设备没有被中断时. 确认返回 IRQ_NONE 无论何时你的处理者被调用并且发现设备没被中断.

如果你需要探测你的设备, 在请求 IRQ 线之前, 内核无法帮你. 没有探测函数可给共享处理者使用. 标准的探测机制有效如果使用的线是空闲的, 但是如果这条线已经被另一个有共享能力的驱动持有, 探测失败, 即便你的驱动已正常工作. 幸运的是, 大部分设计为中断共享的硬件能够告知处理器它在使用哪个中断, 因此减少明显的探测的需要.

释放处理者以正常方式进行, 使用 free_irq. 这里 dev_id 参数用来从这个中断的共享处理者列表中选择正确的处理者来释放. 这就是为什么 dev_id 指针必须是独特的.

一个使用共享处理者的驱动需要小心多一件事: 它不能使用 enable_irq 或者 disable_irq. 如果它用了, 对其他共享这条线的设备就乱了; 禁止另一个设备的中断即便短时间也可能产生延时, 这对这个设备和它的用户是有问题的. 通常, 程序员必须记住, 他的驱动不拥有这个 IRQ, 并且它的行为应当比它拥有这个中断线更加"社会性".

10.4.2. 运行处理者

如同前面建议的, 当内核收到一个中断, 所有的注册的处理者被调用. 一个共享的处理者必须能够在它需要的处理的中断和其他设备产生的中断之间区分.

使用 shared=1 选项来加载 short 安装了下列处理者来代替缺省的:

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int value, written;
        struct timeval tv;
        /* If it wasn't short, return immediately */
        value = inb(short_base);
        if (!(value & 0x80))
                return IRQ_NONE;
        /* clear the interrupting bit */
        outb(value & 0x7F, short_base);
        /* the rest is unchanged */
        do_gettimeofday(&tv);
        written = sprintf((char *)short_head,"%08u.%06u\n",
                          (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
        short_incr_bp(&short_head, written);
        wake_up_interruptible(&short_queue); /* awake any reading process */
        return IRQ_HANDLED;
}
```

这里应该有个解释. 因为并口没有"中断挂起"位来检查, 处理者使用 ACK 位作此目的. 如果这个位是高, 正报告的中断是给 short, 并且这个处理者清除这个位.

处理者通过并口数据端口的清零来复位这个位 -- short 假设管脚 9 和 10 连在一起. 如果其他一个和 short 共享这个 IRQ 的设备产生一个中断, short 看到它的线仍然非激活并且什么不作.

当然, 一个功能齐全的驱动可能将工作划分位前和后半部, 但是容易添加并且不会有任何影响实现共享的代码. 一个真实驱动还可能使用 dev_id 参数来决定, 在很多可能的中, 哪个设备在中断.

注意, 如果你使用打印机(代替跳线)来测试使用 short 的中断管理, 这个共享的处理者不象所说的一样工作,因为打印机协议不允许共享, 并且驱动不知道是否这个中断是来自打印机.

10.4.3. /proc 接口和共享中断

在系统中安装共享处理者不影响 /proc/stat, 它甚至不知道处理者. 但是, /proc/interrupts 稍稍变化.

所有同一个中断号的安装的处理者出现在 /proc/interrupts 的同一行. 下列输出(从一个 x86_64 系统)显示了共享中断处理是如何显示的:

```
 CPU0
 0: 892335412 XT-PIC timer
 1: 453971 XT-PIC i8042
 2: 0 XT-PIC cascade
 5: 0 XT-PIC libata, ehci_hcd
 8: 0 XT-PIC rtc
 9: 0 XT-PIC acpi
 10: 11365067 XT-PIC ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx, EMU10K1
 11: 4391962 XT-PIC uhci_hcd, uhci_hcd
 12: 224 XT-PIC i8042
 14: 2787721 XT-PIC ide0
 15: 203048 XT-PIC ide1
NMI: 41234
LOC: 892193503
ERR: 102
MIS: 0
```

这个系统有几个共享中断线. IRQ 5 用来给串行 ATA 和 IEEE 1394 控制器; IRQ 10 有几个设备, 包括一个 IDE 控制器, 2 个 USB 控制器, 一个以太网接口, 以及一个声卡; 并且 IRQ 11 也被 2 个 USB 控制器使用.

10.5. 中断驱动 I/O

无论何时一个数据传送到或自被管理的硬件可能因为任何原因而延迟, 驱动编写者应当实现缓存. 数据缓存帮助来分离数据传送和接收从写和读系统调用, 并且整个系统性能受益.

一个好的缓存机制产生了中断驱动的 I/O, 一个输入缓存在中断时填充并且被读取设备的进程清空; 一个输出缓存由写设备的进程填充并且在中断时清空. 一个中断驱动的输出的例子是 /dev/shortprint 的实现.

为使中断驱动的数据传送成功发生, 硬件应当能够产生中断, 使用下列语义:

- 对于输入, 设备中断处理器, 当新数据到达时, 并且准备好被系统处理器获取. 进行的实际动作依赖是否设备使用 I/O 端口, 内存映射, 或者 DMA.
- 对于输出, 设备递交一个中断, 或者当它准备好接受新数据, 或者确认一个成功的数据传送. 内存映射的和能 DMA 的设备常常产生中断来告诉系统它们完成了这个缓存.

在一个读或写与实际数据到达之间的时间关系在第 6 章的"阻塞和非阻塞操作"一节中介绍.

10.5.1. 一个写缓存例子

我们已经几次提及 shortprint 驱动; 现在是时候真正看看. 这个模块为并口实现一个非常简单, 面向输出的驱动; 它是足够的, 但是, 来使能文件打印. 如果你选择来测试这个驱动, 但是, 记住你必须传递给打印机一个文件以它理解的格式; 不是所有的打印机在给一个任意数据的流时很好响应.

shortprint 驱动维护一个一页的环形输出缓存. 当一个用户空间进程写数据到这个设备, 数据被填入缓存, 但是写方法实际没有进行任何 I/O. 相反, shortp_write 的核心看来如此:

```
while (written < count)
{
        /* Hang out until some buffer space is available. */
        space = shortp_out_space();
        if (space <= 0) {
                if (wait_event_interruptible(shortp_out_queue,
                                             (space = shortp_out_space()) > 0))
                        goto out;
        }

        /* Move data into the buffer. */
        if ((space + written) > count)
                space = count - written;

        if (copy_from_user((char *) shortp_out_head, buf, space)) {
                up(&shortp_out_sem);
                return -EFAULT;
        }
        shortp_incr_out_bp(&shortp_out_head, space);
        buf += space;
        written += space;

        /* If no output is active, make it active. */
        spin_lock_irqsave(&shortp_out_lock, flags);
        if (! shortp_output_active)
                shortp_start_output();
        spin_unlock_irqrestore(&shortp_out_lock, flags);
}
out:
*f_pos += written;
```

一个旗标 (shortp_out_sem) 控制对这个环形缓存的存取; shortp_write 就在上面的代码片段之前获得这个旗标. 当持有这个旗标, 它试图输入数据到这个环形缓存. 函数 shortp_out_space 返回可用的连续空间的数量(因此, 没有必要担心缓存回绕); 如果这个量是 0, 驱动等到释放一些空间. 它接着拷贝它能够的数量的数据到缓存中.

一旦有数据输出, shortp_write 必须确保数据被写到设备. 数据的写是通过一个工作队列函数完成的; shortp_write 必须启动这个函数如果它还未在运行. 在获取了一个单独

的，控制存取输出缓存的消费者一侧(包括 shortp_output_active)的数据的自旋锁后，它调用 shortp_start_output 如果需要．接着只是注意多少数据被写到缓存并且返回．

启动输出进程的函数看来如下：

```
static void shortp_start_output(void)
{
        if (shortp_output_active) /* Should never happen */
                return;

        /* Set up our 'missed interrupt' timer */
        shortp_output_active = 1;
        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);

        /* And get the process going. */
        queue_work(shortp_workqueue, &shortp_work);
}
```

处理硬件的事实是，你可以，偶尔，丢失来自设备的中断．当发生这个，你确实不想你的驱动一直停止直到系统重启；这不是一个用户友好的做事方式．最好是认识到一个中断已经丢失，收拾残局，继续．为此，shortprint 甚至一个内核定时器无论何时它输出数据给设备．如果时钟超时，我们可能丢失一个中断．我们很快会看到定时器函数，但是，暂时，让我们坚持在主输出功能上．那是在我们的工作队列函数里实现的，它，如同你上面看到的，在这里被调度．那个函数的核心看来如下：

```
spin_lock_irqsave(&shortp_out_lock, flags);
/* Have we written everything? */
if (shortp_out_head == shortp_out_tail)
{ /* empty */
        shortp_output_active = 0;
        wake_up_interruptible(&shortp_empty_queue);
        del_timer(&shortp_timer);
}
/* Nope, write another byte */
else
        shortp_do_write();
/* If somebody's waiting, maybe wake them up. */
if (((PAGE_SIZE + shortp_out_tail -shortp_out_head) % PAGE_SIZE) > SP_MIN_SPACE)
{
        wake_up_interruptible(&shortp_out_queue);
}
spin_unlock_irqrestore(&shortp_out_lock, flags);
```

因为我们在使用共享变量的输出一侧，我们必须获得自旋锁．接着我们看是否有更多的数据要发送；如果无，我们注意输出不再激活，删除定时器，并且唤醒任何在等待队列全空的进程(这种等待当设备被关闭时结束)．如果，相反，有数据要写，我们调用 shortp_do_write 来实际发送一个字节到硬件．

接着，因为我们可能在输出缓存中有空闲空间，我们考虑唤醒任何等待增加更多数据给那个缓存的进程．但是我们不是无条件进行唤醒；相反，我们等到有一个最低数量的空间．

每次我们从缓存拿出一个字节就唤醒一个写者是无意义的; 唤醒进程的代价, 调度它运行, 并且使它重回睡眠, 太高了. 相反, 我们应当等到进程能够立刻移动相当数量的数据到缓存. 这个技术在缓存的, 中断驱动的驱动中是普通的.

为完整起见, 这是实际写数据到端口的代码:

```
static void shortp_do_write(void)
{
        unsigned char cr = inb(shortp_base + SP_CONTROL);
        /* Something happened; reset the timer */
        mod_timer(&shortp_timer, jiffies + TIMEOUT);
        /* Strobe a byte out to the device */
        outb_p(*shortp_out_tail, shortp_base+SP_DATA);
        shortp_incr_out_bp(&shortp_out_tail, 1);
        if (shortp_delay)
                udelay(shortp_delay);
        outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
        if (shortp_delay)
                udelay(shortp_delay);
        outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

这里, 我们复位定时器来反映一个事实, 我们已经作了一些处理, 输送字节到设备, 并且更新了环形缓存指针.

工作队列函数没有直接重新提交它自己, 因此只有一个单个字节会被写入设备. 在某一处, 打印机将, 以它的缓慢方式, 消耗这个字节并且准备好下一个; 它将接着中断处理器. shortprint 中使用的中断处理是简短的:

```
static irqreturn_t shortp_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        if (! shortp_output_active)
                return IRQ_NONE;
        /* Remember the time, and farm off the rest to the workqueue function */
        do_gettimeofday(&shortp_tv);
        queue_work(shortp_workqueue, &shortp_work);
        return IRQ_HANDLED;
}
```

因为并口不要求一个明显的中断确认, 中断处理所有真正需要做的是告知内核来再次运行工作队列函数.

如果中断永远不来如何? 至此我们已见到的驱动代码将简单地停止. 为避免发生这个, 我们设置了一个定时器在几页前. 当定时器超时运行的函数是:

```
static void shortp_timeout(unsigned long unused)
{
        unsigned long flags;
        unsigned char status;
        if (! shortp_output_active)
                return;
```

```
        spin_lock_irqsave(&shortp_out_lock, flags);
        status = inb(shortp_base + SP_STATUS);

        /* If the printer is still busy we just reset the timer */
        if ((status & SP_SR_BUSY) == 0 || (status & SP_SR_ACK)) {

                shortp_timer.expires = jiffies + TIMEOUT;
                add_timer(&shortp_timer);
                spin_unlock_irqrestore(&shortp_out_lock, flags);
                return;
        }
        /* Otherwise we must have dropped an interrupt. */
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        shortp_interrupt(shortp_irq, NULL, NULL);
}
```

如果没有输出要被激活，定时器函数简单地返回. 这避免了定时器重新提交自己，当事情在被关闭时. 接着，在获得了锁之后，我们查询端口的状态；如果它声称忙，它完全还没有时间来中断我们，因此我们复位定时器并且返回. 打印机能够，有时，花很长时间来使自己准备；考虑一下缺纸的打印机，而每个人在一个长周末都不在. 在这种情况下，只有耐心等待直到事情改变.

但是，如果打印机声称准备好了，我们一定丢失了它的中断. 这个情况下，我们简单地手动调用我们的中断处理来使输出处理再动起来.

shortpirnt 驱动不支持从端口读数据；相反，它象 shortint 并且返回中断时间信息. 但是一个中断驱动的读方法的实现可能非常类似我们已经见到的. 从设备来的数据可能被读入驱动缓存；它可能被拷贝到用户空间只在缓存中已经累积了相当数量的数据，完整的读请求已被满足，或者某种超时发生.

10.6. 快速参考

本章中介绍了这些关于中断管理的符号：

```
#include <linux/interrupt.h>
int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long flags, const
char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

调用这个注册和注销一个中断处理.

```
#include <linux/irq.h.h>
int can_request_irq(unsigned int irq, unsigned long flags);
```

这个函数，在 i386 和 x86_64 体系上有，返回一个非零值如果一个分配给定中断线的企图成功.

```
#include <asm/signal.h>
SA_INTERRUPT
SA_SHIRQ
```

SA_SAMPLE_RANDOM

给 request_irq 的标志. SA_INTERRUPT 请求安装一个快速处理者(相反是一个慢速的). SA_SHIRQ 安装一个共享的处理者, 并且第 3 个 flag 声称中断时戳可用来产生系统熵.

/proc/interrupts
/proc/stat

报告硬件中断和安装的处理者的文件系统节点.

```
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
```

驱动使用的函数, 当它不得不探测来决定哪个中断线被设备在使用. probe_irq_on 的结果必须传回给 probe_irq_off 在中断产生之后. probe_irq_off 的返回值是被探测的中断号.

IRQ_NONE
IRQ_HANDLED
IRQ_RETVAL(int x)

从一个中断处理返回的可能值, 指示是否一个来自设备的真正的中断出现了.

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

驱动可以使能和禁止中断报告. 如果硬件试图在中断禁止时产生一个中断, 这个中断永远丢失了. 一个使用一个共享处理者的驱动必须不使用这个函数.

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

使用 local_irq_save 来禁止本地处理器的中断并且记住它们之前的状态. flags 可以被传递给 local_irq_restore 来恢复之前的中断状态.

```
void local_irq_disable(void);
void local_irq_enable(void);
```

在当前处理器熵无条件禁止和使能中断的函数.

第 11 章 内核中的数据类型

在我们进入更高级主题之前，我们需要停下来快速关注一下可移植性问题. 现代版本的 Linux 内核是高度可移植的，它正运行在很多不同体系上. 由于 Linux 内核的多平台特性，打算做认真使用的驱动应当也是可移植的.

但是内核代码的一个核心问题是不但能够存取已知长度的数据项(例如，文件系统数据结构或者设备单板上的寄存器)，而且可以使用不同处理器的能力(32-位 和 64-位 体系，并且也可能是 16 位).

内核开发者在移植 x86 代码到新体系时遇到的几个问题与不正确的数据类型相关. 坚持严格的数据类型和使用 -Wall -Wstrict-prototypes 进行编译可能避免大部分的 bug.

内核数据使用的数据类型分为 3 个主要类型：标准 C 类型例如 int，明确大小的类型例如 u32，以及用作特定内核对象的类型，例如 pid_t. 我们将看到这 3 个类型种类应当什么时候以及应当如何使用. 本章的最后的节谈论一些其他的典型问题，你在移植 x86 的驱动到其他平台时可能遇到的问题，并且介绍近期内核头文件输出的链表的常用支持.

如果你遵照我们提供的指引，你的驱动应当编译和运行在你无法测试的平台上.

11.1. 标准 C 类型的使用

尽管大部分程序员习惯自由使用标准类型，如 int 和 long，编写设备驱动需要一些小心来避免类型冲突和模糊的 bug.

这个问题是你不能使用标准类型，当你需要"一个 2-字节 填充者"或者"一个东西来代表一个 4-字节 字串"，因为正常的 C 数据类型在所有体系上不是相同大小. 为展示各种 C 类型的数据大小，datasize 程序已包含在例子文件 misc-progs 目录中，由 O' Reilly's FTP 站点提供. 这是一个程序的样例运行，在一个 i386 系统上(显示的最后 4 个类型在下一章介绍)：

```
morgana% misc-progs/datasize
arch Size: char short int long ptr long-long u8 u16 u32 u64
i686       1     2     4   4   4   8          1  2   4   8
```

这个程序可以用来显示长整型和指针在 64-位 平台上的不同大小，如同在不同 Linux 计算机上运行程序所演示的：

```
arch   Size:  char  short  int  long  ptr long-long  u8 u16 u32 u64
i386          1     2      4    4     4   8          1  2    4   8
alpha         1     2      4    8     8   8          1  2    4   8
armv4l        1     2      4    4     4   8          1  2    4   8
ia64          1     2      4    8     8   8          1  2    4   8
m68k          1     2      4    4     4   8          1  2    4   8
mips          1     2      4    4     4   8          1  2    4   8
ppc           1     2      4    4     4   8          1  2    4   8
sparc         1     2      4    4     4   8          1  2    4   8
sparc64       1     2      4    4     4   8          1  2    4   8
x86_64        1     2      4    8     8   8          1  2    4   8
```

注意有趣的是 SPARC 64 体系在一个 32-位 用户空间运行, 因此那里指针是 32 位宽, 尽管它们在内核空间是 64 位宽. 这可用加载 kdatasize 模块(在例子文件的 misc-modules 目录里)来验证. 这个模块在加载时使用 printk 来报告大小信息, 并且返回一个错误(因此没有必要卸载它):

```
kernel: arch Size: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64     1     2     4    8    8    8          1   2    4    8
```

尽管在混合不同数据类型时你必须小心, 有时有很好的理由这样做. 一种情况是因为内存存取, 与内核相关时是特殊的. 概念上, 尽管地址是指针, 内存管理常常使用一个无符号的整数类型更好地完成; 内核对待物理内存如同一个大数组, 并且内存地址只是一个数组索引. 进一步地, 一个指针容易解引用; 当直接处理内存存取时, 你几乎从不想以这种方式解引用. 使用一个整数类型避免了这种解引用, 因此避免了 bug. 因此, 内核中通常的内存地址常常是 unsigned long, 利用了指针和长整型一直是相同大小的这个事实, 至少在 Linux 目前支持的所有平台上.

因为其所值的原因, C99 标准定义了 intptr_t 和 uintptr_t 类型给一个可以持有一个指针值的整型变量. 但是, 这些类型几乎没在 2.6 内核中使用.

11.2. 安排一个明确大小给数据项

有时内核代码需要一个特定大小的数据项, 也许要匹配预定义的二进制结构,[39] 来和用户空间通讯, 或者来用插入"填充"字段来对齐结构中的数据(但是关于对齐问题的信息参考"数据对齐" 一节).

内核提供了下列数据类型来使用, 无论你什么时候需要知道你的数据的大小. 所有的数据声明在 <asm/types.h>, 它又被 <linux/types.h> 包含.

```
u8; /* unsigned byte (8 bits) */
u16; /* unsigned word (16 bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
```

存在对应的有符号类型, 但是很少需要; 如果你需要它们, 只要在名子里用 s 代替 u.

如果一个用户空间程序需要使用这些类型, 可用使用一个双下划线前缀在名子上: __u8 和其它独立于 __KERNEL__ 定义的类型. 例如, 如果, 一个驱动需要与用户空间中运行的程序交换二进制结构, 通过 ioctl, 头文件应当在结构中声明 32-位 成员为 __u32.

重要的是记住这些类型是 Linux 特定的, 并且使用它们妨碍了移植软件到其他的 Unix 口味上. 使用近期编译器的系统支持 C99-标准 类型, 例如 uint8_t 和 uint32_t; 如果考虑到移植性, 使用这些类型比 Linux-特定的变体要好.

你可能也注意到有时内核使用传统的类型, 例如 unsigned int, 给那些维数与体系无关的项. 这是为后向兼容而做的. 当 u32 和它的类似物在版本 1.1.67 引入时, 开发者不能改变存在的数据结构为新的类型, 因为编译器发出一个警告当在结构成员和安排给它的

值之间有一个类型不匹配时.. Linus 不希望他写给自己使用的操作系统称为多平台的; 结果是, 老的结构有时被松散的键入.

事实上, 编译器指示类型不一致, 甚至在 2 个类型只是同一个对象的不同名子, 例如在 PC 上 unsigned long 和 u32.

---

[39] 这发生在当读取分区表时, 当执行一个二进制文件时, 或者当解码一个网络报文时.

11.3. 接口特定的类型

内核中一些通常使用的数据类型有它们自己的 typedef 语句, 因此阻止了任何移植性问题. 例如, 一个进程标识符 (pid) 常常是 pid_t 而不是 int. 使用 pid_t 屏蔽了任何在实际数据类型上的不同. 我们使用接口特定的表达式来指一个类型, 由一个库定义的, 以便于提供一个接口给一个特定的数据结构.

注意, 在近期, 已经相对少定义新的接口特定类型. 使用 typedef 语句已经有许多内核开发者不喜欢, 它们宁愿看到代码中直接使用的真实类型信息, 不是藏在一个用户定义的类型后面. 很多老的接口特定的类型在内核中保留, 但是, 并且它们不会很快消失.

甚至当没有定义接口特定的类型, 以和内核其他部分保持一致的方式使用正确的数据类型是一直重要的. 一个嘀哒计数, 例如, 一直是 unsigned long, 独立于它实际的大小, 因此 unsigned long 类型应当在使用 jiffy 时一直使用. 本节我们集中于 _t 类型的使用.

很多 _t 类型在 <linux/types.h> 中定义, 但是列出来是很少有用. 当你需要一个特定类型, 你会在你需要调用的函数的原型中发现它, 或者在你使用的数据结构中.

无论何时你的驱动使用需要这样"定制"类型的函数并且你不遵照惯例, 编译器发出一个警告; 如果你使用 -Wall 编译器标志并且小心去除所有的警告, 你能有信心你的代码是可移植的.

_t 数据项的主要问题是当你需要打印它们时, 常常不容易选择正确的 printk 或 printf 格式, 你在一个体系上出现的警告会在另一个上重新出现. 例如, 你如何打印一个 size_t, 它在一些平台上是 unsigned long 而在其他某个上面是 unsigned int?

无论何时你需要打印某个接口特定的数据, 最好的方法是转换它的值为最大的可能类型(常常是 long 或者 unsigned long) 并且接着打印它通过对应的格式. 这种调整不会产生错误或者警告, 因为格式匹配类型, 并且你不会丢失数据位, 因为这个转换或者是一个空操作或者是数据项向更大数据类型的扩展.

实际上, 我们在谈论的数据项不会常常要打印的, 因此这个问题只适用于调试信息. 常常, 代码只需要存储和比较接口特定的类型, 加上传递它们作为给库或者内核函数的参数.

尽管 _t 类型是大部分情况的正确解决方法, 有时正确的类型不存取. 这发生在某些还未被清理的老接口.

我们在内核头文件中发现的一个模糊之处是用在 I/O 函数的数据类型，它松散地定义 (看第 9 章"平台相关性" 一节). 松散的类型在那里主要是因为历史原因，但是在写代码时它可能产生问题. 例如，交换给函数如 outb 的参数可能会有麻烦；如果有一个 port_t 类型，编译器会发现这个类型.

11.4. 其他移植性问题

除了数据类型，当编写一个驱动时有几个其他的软件问题要记住，如果你想在 Linux 平台间可移植.

一个通常的规则是怀疑显式的常量值. 常常通过使用预处理宏，代码已被参数化. 这一节列出了最重要的可移植性问题. 无论何时你遇到已被参数化的值，你可以在头文件中以及在随官方内核发布的设备驱动中找到提示.

11.4.1. 时间间隔

当涉及时间间隔，不要假定每秒有 1000 个嘀哒. 尽管当前对 i386 体系是真实的，不是每个 Linux 平台都以这个速度运行. 对于 x86 如果你使用 HZ 值(如同某些人做的那样)，这个假设可能是错的，并且没人知道将来内核会发生什么. 无论何时你使用嘀哒来计算时间间隔，使用 HZ (每秒的定时器中断数) 来标定你的时间. 例如，检查一个半秒的超时，用 HZ/2 和逝去时间比较. 更普遍地，msec 毫秒对应地嘀哒数一直是 msec*HZ/1000.

11.4.2. 页大小

当使用内存时，记住一个内存页是 PAGE_SIZE 字节，不是 4KB. 假定页大小是 4KB 并且硬编码这个值是一个 PC 程序员常见的错误，相反，被支持的平台显示页大小从 4 KB 到 64 KB，并且有时它们在相同平台上的不同的实现上不同. 相关的宏定义是 PAGE_SIZE 和 PAGE_SHIT. 后者包含将一个地址移位来获得它的页号的位数. 对于 4KB 或者更大的页这个数当前是 12 或者更大. 宏在 <asm/page.h> 中定义；用户空间程序可以使用 getpagesize 库函数，如果它们需要这个信息.

让我们看一下非一般的情况. 如果一个驱动需要 16 KB 来暂存数据，它不应当指定一个 2 的指数 给 get_free_pages. 你需要一个可移植解决方法. 这样的解决方法，幸运的是，已经由内核开发者写好并且称为 get_order:

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

记住，get_order 的参数必须是 2 的幂.

11.4.3. 字节序

小心不要假设字节序. PC 存储多字节值是低字节为先(小端为先，因此是小端)，一些高级的平台以另一种方式(大端)工作. 任何可能的时候，你的代码应当这样来编写，它不在乎它操作的数据的字节序. 但是，有时候一个驱动需要使用单个字节建立一个整型数或者相反，或者它必须与一个要求一个特定顺序的设备通讯.

包含文件 <asm/byteorder.h> 定义了或者 __BIG_ENDIAN 或者 __LITTLE_ENDIAN, 依赖处理器的字节序. 当处理字节序问题时, 你可能编码一堆 #ifdef __LITTTLE_ENDIAN 条件语句, 但是有一个更好的方法. Linux 内核定义了一套宏定义来处理之间的转换, 在处理器字节序和你需要以特定字节序存储和加载的数据之间. 例如:

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

这 2 个宏定义转换一个值, 从无论 CPU 使用的什么到一个无符号的, 小端, 32 位数, 并且转换回. 它们不管你的 CPU 是小端还是大端, 不管它是不是 32-位 处理器. 在没有事情要做的情况下它们原样返回它们的参数. 使用这些宏定义易于编写可移植的代码, 而不必使用大量的条件编译建造.

有很多类似的函数; 你可以在 <linux/byteorder/big_endian.h> 和 <linux/byteorder/little_endian.h> 中见到完整列表. 一会儿之后, 这个模式不难遵循. be64_to_cpu 转换一个无符号的, 大端, 64-位 值到一个内部 CPU 表示. le16_to_cpus, 相反, 处理有符号的, 小端, 16 位数. 当处理指针时, 你也会使用如 cpu_to_le32p, 它使用指向一个值的指针来转换, 而不是这个值自身. 剩下的看包含文件.

11.4.4. 数据对齐

编写可移植代码而值得考虑的最后一个问题是如何存取不对齐的数据 -- 例如, 如何读取一个存储于一个不是 4 字节倍数的地址的 4 字节值. i386 用户常常存取不对齐数据项, 但是不是所有的体系允许这个. 很多现代的体系产生一个异常, 每次程序试图不对齐数据传送时; 数据传输由异常处理来处理, 带来很大的性能牺牲. 如果你需要存取不对齐的数据, 你应当使用下列宏:

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是无类型的, 并且用在每个数据项, 不管它是 1 个, 2 个, 4 个, 或者 8 个字节长. 它们在任何内核版本中定义.

关于对齐的另一个问题是跨平台的数据结构移植性. 同样的数据结构(在 C-语言 源文件中定义)可能在不同的平台上不同地编译. 编译器根据各个平台不同的惯例来安排结构成员对齐.

为了编写可以跨体系移动的数据使用的数据结构, 你应当一直强制自然的数据项对齐, 加上对一个特定对齐方式的标准化. 自然对齐意味着存储数据项在是它的大小的整数倍的地址上(例如, 8-byte 项在 8 的整数倍的地址上). 为强制自然对齐在阻止编译器以不希望的方式安排成员量的时候, 你应当使用填充者成员来避免在数据结构中留下空洞.

为展示编译器如何强制对齐, dataalign 程序在源码的 misc-progs 目录中发布, 并且一个对等的 kdataalign 模块是 misc-modules 的一部分. 这是程序在几个平台上的输出以及模块在 SPARC64 的输出:

```
arch Align: char short int long ptr long-long u8 u16 u32 u64
i386        1     2     4   4    4   4         1  2   4   4
i686        1     2     4   4    4   4         1  2   4   4
alpha       1     2     4   8    8   8         1  2   4   8
armv4l      1     2     4   4    4   4         1  2   4   4
ia64        1     2     4   8    8   8         1  2   4   8
mips        1     2     4   4    4   8         1  2   4   8
ppc         1     2     4   4    4   8         1  2   4   8
sparc       1     2     4   4    4   8         1  2   4   8
sparc64     1     2     4   4    4   8         1  2   4   8
x86_64      1     2     4   8    8   8         1  2   4   8

kernel: arch Align: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64     1     2     4   8    8   8         1  2   4   8
```

有趣的是注意不是所有的平台对齐 64-位值在 64-位边界上，因此你需要填充者成员来强制对齐和保证可移植性.

最后，要知道编译器可能自己悄悄地插入填充到结构中来保证每个成员是对齐的，为了目标处理器的良好性能. 如果你定义一个结构打算来匹配一个设备期望的结构，这个自动的填充可能妨碍你的企图. 解决这个问题的方法是告诉编译器这个结构必须是"紧凑的"，不能增加填充者. 例如，内核头文件 <linux/edd.h> 定义几个与 x86 BIOS 接口的数据结构，并且它包含下列的定义:

```
struct
{
        u16 id;
        u64 lun;
        u16 reserved1;
        u32 reserved2;
}
__attribute__ ((packed)) scsi;
```

如果没有 __attribute__ ((packed))，lun 成员可能被在前面添加 2 个填充者字节或者 6 个，如果我们在 64-位平台上编译这个结构.

11.4.5. 指针和错误值

很多内部内核函数返回一个指针值给调用者. 许多这些函数也可能失败. 大部分情况，失败由返回一个 NULL 指针值来指示. 这个技术是能用的，但是它不能通知问题的确切特性. 一些接口确实需要返回一个实际的错误码以便于调用者能够基于实际上什么出错来作出正确的判断.

许多内核接口通过在指针值中对错误值编码来返回这个信息. 这样的信息必须小心使用，因为它们的返回值不能简单地与 NULL 比较. 为帮助创建和使用这类接口，一小部分函数已可用(在 <linux/err.h>).

一个返回指针类型的函数可以返回一个错误值，使用：

```
void *ERR_PTR(long error);
```

这里，error 是常见的负值错误码. 调用者可用使用 IS_ERR 来测试是否一个返回的指针是不是一个错误码：

```
long IS_ERR(const void *ptr);
```

如果你需要实际的错误码，它可能被抽取到，使用：

```
long PTR_ERR(const void *ptr);
```

你应当只对 IS_ERR 返回一个真值的值使用 PTR_ERR；任何其他的值是一个有效的指针.

11.5. 链表

操作系统内核，如同其他程序，常常需要维护数据结构的列表. 有时，Linux 内核已经同时有几个列表实现. 为减少复制代码的数量，内核开发者已经创建了一个标准环形的，双链表；鼓励需要操作列表的人使用这个设施.

当使用链表接口时，你应当一直记住列表函数不做加锁. 如果你的驱动可能试图对同一个列表并发操作，你有责任实现一个加锁方案. 可选项(破坏的列表结构，数据丢失，内核崩溃) 肯定是难以诊断的.

为使用列表机制，你的驱动必须包含文件 <linux/list.h>. 这个文件定义了一个简单的类型 list_head 结构：

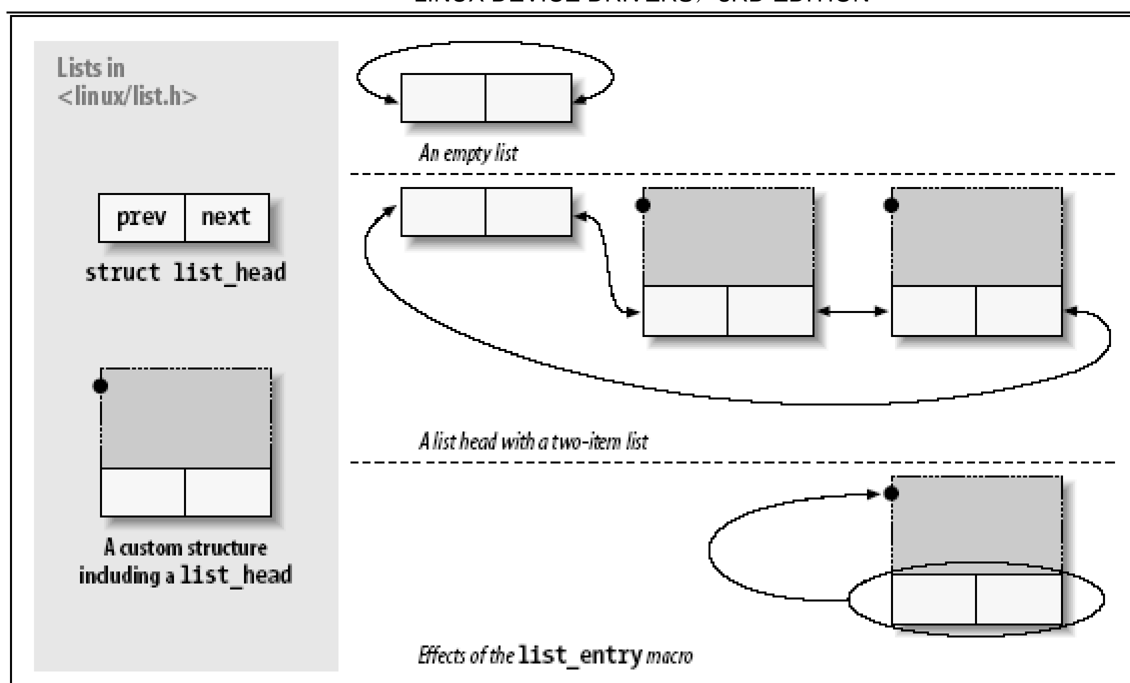
```
struct list_head { struct list_head *next, *prev; };
```

真实代码中使用的链表几乎是不变地由几个结构类型组成，每一个描述一个链表中的入口项. 为在你的代码中使用 Linux 列表，你只需要嵌入一个 list_head 在构成这个链表的结构里面. 假设，如果你的驱动维护一个列表，它的声明可能看起来象这样：

```
struct todo_struct
{
        struct list_head list;
        int priority; /* driver specific */
        /* ... add other driver-specific fields */
};
```

列表的头常常是一个独立的 list_head 结构. 图链表头数据结构显示了这个简单的 struct list_head 是如何用来维护一个数据结构的列表的.

图 11.1. 链表头数据结构



链表头必须在使用前用 INIT_LIST_HEAD 宏来初始化. 一个"要做的事情"的链表头可能声明并且初始化用:

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
<para>可选地, 链表可在编译时初始化:</para>
LIST_HEAD(todo_list);
```

几个使用链表的函数定义在 <linux/list.h>:

```
list_add(struct list_head *new, struct list_head *head);
```

在紧接着链表 head 后面增加新入口项 -- 正常地在链表的开头. 因此, 它可用来构建堆栈. 但是, 注意, head 不需要是链表名义上的头; 如果你传递一个 list_head 结构, 它在链表某处的中间, 新的项紧靠在它后面. 因为 Linux 链表是环形的, 链表的头通常和任何其他的项没有区别.

```
list_add_tail(struct list_head *new, struct list_head *head);
```

刚好在给定链表头前面增加一个新入口项 -- 在链表的尾部, 换句话说. list_add_tail 能够, 因此, 用来构建先入先出队列.

```
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
```

给定的项从队列中去除. 如果入口项可能注册在另外的链表中, 你应当使用 list_del_init, 它重新初始化这个链表指针.

```
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
```

给定的入口项从它当前的链表里去除并且增加到 head 的开始. 为安放入口项在新链表的末尾, 使用 list_move_tail 代替.

list_empty(struct list_head *head);

如果给定链表是空, 返回一个非零值.

list_splice(struct list_head *list, struct list_head *head);

将 list 紧接在 head 之后来连接 2 个链表.

list_head 结构对于实现一个相似结构的链表是好的, 但是调用程序常常感兴趣更大的结构, 它组成链表作为一个整体. 一个宏定义, list_entry, 映射一个 list_head 结构指针到一个指向包含它的结构的指针. 它如下被调用:

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

这里 ptr 是一个指向使用的 struct list_head 的指针, type_of_struct 是包含 ptr 的结构的类型, field_name 是结构中列表成员的名子. 在我们之前的 todo_struct 结构中, 链表成员称为简单列表. 因此, 我们应当转变一个列表入口项为它的包含结构, 使用这样一行:

```
struct todo_struct *todo_ptr = list_entry(listptr, struct todo_struct, list);
```

list_entry 宏定义使用了一些习惯的东西但是不难用.

链表的遍历是容易的: 只要跟随 prev 和 next 指针. 作为一个例子, 假设我们想保持 todo_struct 项的列表已降序的优先级顺序排列. 一个函数来添加新项应当看来如此:

```
void todo_add_entry(struct todo_struct *new)
{
        struct list_head *ptr;
        struct todo_struct *entry;

        for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next)
        {
                entry = list_entry(ptr, struct todo_struct, list);
                if (entry->priority < new->priority) {

                        list_add_tail(&new->list, ptr);
                        return;
                }
        }
        list_add_tail(&new->list, &todo_struct)

}
```

但是，作为一个通用的规则，最好使用一个预定义的宏来创建循环，它遍历链表. 前一个循环，例如，可这样编码:

```
void todo_add_entry(struct todo_struct *new)
{
        struct list_head *ptr;
        struct todo_struct *entry;

        list_for_each(ptr, &todo_list)
        {
                entry = list_entry(ptr, struct todo_struct, list);
                if (entry->priority < new->priority) {

                        list_add_tail(&new->list, ptr);
                        return;
                }
        }
        list_add_tail(&new->list, &todo_struct)

}
```

使用提供的宏帮助避免简单的编程错误；宏的开发者也已做了些努力来保证它们进行正常. 存在几个变体:

list_for_each(struct list_head *cursor, struct list_head *list)

这个宏创建一个 for 循环，执行一次，cursor 指向链表中的每个连续的入口项. 小心改变列表在遍历它时.

list_for_each_prev(struct list_head *cursor, struct list_head *list)

这个版本后向遍历链表.

list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)

如果你的循环可能删除列表中的项，使用这个版本. 它简单的存储列表 next 中下一个项，在循环的开始，因此如果 cursor 指向的入口项被删除，它不会被搞乱.

list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)

这些宏定义减轻了对一个包含给定结构类型的列表的处理. 这里，cursor 是一个指向包含数据类型的指针，member 是包含结构中 list_head 结构的名子. 有了这些宏，没有必要安放 list_entry 调用在循环里了.

如果你查看 <linux/list.h> 里面, 你看到有一些附加的声明. hlist 类型是一个有一个单独的, 单指针列表头类型的双向链表; 它常用作创建哈希表和类型结构. 还有宏用来遍历 2 种列表类型, 打算作使用 读取-拷贝-更新 机制(在第 5 章的"读取-拷贝-更新"一节中描述). 这些原语在设备驱动中不可能有用; 看头文件如果你愿意知道更多信息关于它们是如何工作的.

11.6. 快速参考

下列符号在本章中介绍了:

```
#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;
```

保证是 8-位, 16-位, 32-位 和 64-位 无符号整型值的类型. 对等的有符号类型也存在. 在用户空间, 你可用 __u8, __u16, 等等来引用这些类型.

```
#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
```

给当前体系定义每页的字节数, 以及页偏移的位数(对于 4 KB 页是 12, 8 KB 是 13)的符号.

```
#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
```

这 2 个符号只有一个定义, 依赖体系.

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

在已知字节序和处理器字节序之间转换的函数. 有超过 60 个这样的函数: 在 include/linux/byteorder/ 中的各种文件有完整的列表和它们以何种方式定义.

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

一些体系需要使用这些宏保护不对齐的数据存取. 这些宏定义扩展成通常的指针解引用, 为那些允许你存取不对齐数据的体系.

```
#include <linux/err.h>
```

```
void *ERR_PTR(long error);
long PTR_ERR(const void *ptr);
long IS_ERR(const void *ptr);
```

允许错误码由返回指针值的函数返回.

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_splice(struct list_head *list, struct list_head *head);
```

操作环形, 双向链表的函数.

```
list_for_each(struct list_head *cursor, struct list_head *list)
list_for_each_prev(struct list_head *cursor, struct list_head *list)
list_for_each_safe(struct list_head *cursor, struct list_head *next, struct
list_head *list)
list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next struct list_head *list,
member)
```

方便的宏定义, 用在遍历链表上.

第 12 章 PCI 驱动

虽然第 9 章介绍了硬件控制的最低层，本章提供了总线结构的高级一些的概括. 一个总线由电路接口和一个编程接口组成. 在本章，我们涉及编程接口.

本章涉及许多总线结构. 但是，主要的焦点在存取 PCI 外设的内核函数，因为如今 PCI 总线是在桌面计算机和更大的计算机上最普遍使用的外设总线. 这个总线是被内核支持得最好的. ISA 对于电子爱好者仍然是普遍的，在后面描述它，尽管它更多的是一个裸露金属类型的总线，并且没有更多的要讲的，除了在第 9 章和第 10 章涵盖到的.

12.1. PCI 接口

尽管许多计算机用户认为 PCI 是一种电路布线方法，实际上它是一套完整的规格，定义了一个计算机的不同部分应当如何交互.

PCI 规范涉及和计算机接口相关的大部分问题. 我们不会在这里涵盖全部；在本节，我们主要关注一个 PCI 驱动如何能找到它的硬件并获得对它的存取. 在第 2 章的"模块参数"一节和第 10 章的"自动探测 IRQ 号"一节讨论的探测技术可被用在 PCI 设备，但是这个规范提供了一个更适合探测的选择.

PCI 体系被设计为 ISA 标准的替代品，有 3 个主要目的：当在计算机和它的外设之间传送数据时获得更好的性能，尽可能平台无关，以及简化添加和去除系统的外设.

PCI 总线通过使用一个比 ISA 更高的时钟频率，获得更好的性能；它的设置运行在 25 或者 33 MHz (它的实际频率是系统时钟的一个因数)，以及 66-MHz 甚至 133-MHz 的实现最近也已经被采用. 但是，它配备有 32-位 数据线，并且一个 64-位扩展已经被包含在规范中. 平台独立性常常是一个计算机总线设计的目标，并且它是 PCI 的一个特别重要的特性，因为 PC 世界已一直被处理器特定的接口标准占据. PCI 当前广泛用在 IA-32, Alpha，PowerPC，SPARC64，和 IA-64 系统中，以及一些其他的平台.

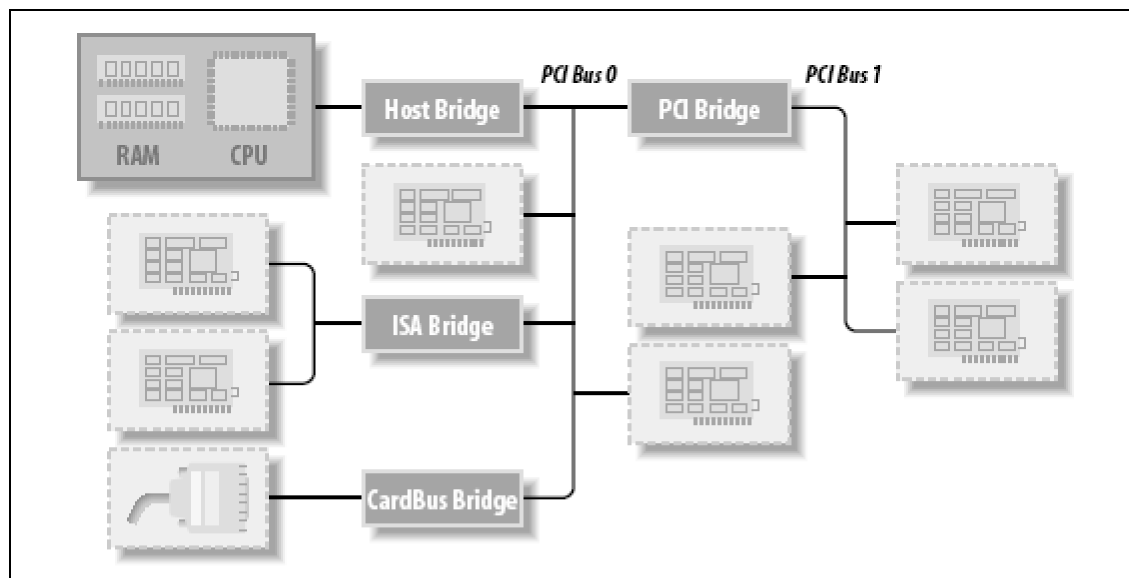
但是，和驱动作者最相关的，是 PCI 对接口板的自动探测的支持. PCI 设备是无跳线的(不象大部分的老式外设)并且是在启动时自动配置的. 接着，设备驱动必须能够存取设备中的配置信息以便能完成初始化. 这不用进行任何探测.

12.1.1. PCI 寻址

每个 PCI 外设有一个总线号，一个设备号，一个功能号标识. PCI 规范允许单个系统占用多达 256 个总线，但是因为 256 个总线对许多大系统是不够的，Linux 现在支持 PCI 域. 每个 PCI 域可以占用多达 256 个总线. 每个总线占用 32 个设备，每个设备可以是一个多功能卡(例如一个声音设备，带有一个附加的 CD-ROM 驱动)有最多 8 个功能. 因此，每个功能可在硬件层次被一个 16-位地址或者 key ，标识. Linux 的设备驱动编写者，然而，不需要处理这些二进制地址，因为它们使用一个特定的数据结构，称为 pci_dev，来在设备上操作.

大部分近期的工作站至少有 2 个 PCI 总线. 在单个系统插入多于 1 个总线要通过桥实现, 桥是特殊用途的 PCI 外设, 它的工作是连接 2 个总线. 一个 PCI 系统的全部分布是一个树, 这里每个总线都连接到一个上层总线, 直到在树根的总线 0 . CardBus PC-card 系统也通过桥连接到 PCI 系统. 图一个典型 PCI 系统的布局表示了一个典型的 PCI 系统, 其中各种桥被突出表示了.

图 12.1. 一个典型 PCI 系统的布局



和 PCI 外设相关的 16-位硬件地址, 尽管大部分隐藏在 struct pci_dev 结构中, 仍然是可偶尔见到, 特别是当使用设备列表. 一个这样的情形是 lspci 的输出(pciutils 的一部分, 在大部分发布中都有)和在 /proc/pci 和 /porc/bus/pci 中的信息排布. PCI 设备的 sysfs 表示也显示了这种寻址方案, 还有 PCI 域信息. [40]当显示硬件地址时, 它可被显示为 2 个值(一个 8-位总线号和一个 8-位 设备和功能号), 作为 3 个值(bus, device, 和 function), 或者作为 4 个值(domain, bus, device, 和 function); 所有的值常常用 16 进制显示.

例如, /proc/bus/pci/devices 使用一个单个 16-位 字段(来便于分析和排序), 而 /proc/bus/busnumber 划分地址为 3 个字段. 下面内容显示了这些地址如何显示, 只显示了输出行的开始:

```
$ lspci | cut -d: -f1-3
0000:00:00.0 Host bridge
0000:00:00.1 RAM memory
0000:00:00.2 RAM memory
0000:00:02.0 USB Controller
0000:00:04.0 Multimedia audio controller
0000:00:06.0 Bridge
0000:00:07.0 ISA bridge
0000:00:09.0 USB Controller
0000:00:09.1 USB Controller
0000:00:09.2 USB Controller
0000:00:0c.0 CardBus bridge
0000:00:0f.0 IDE interface
0000:00:10.0 Ethernet controller
```

```
0000:00:12.0 Network controller
0000:00:13.0 FireWire (IEEE 1394)
0000:00:14.0 VGA compatible controller
$ cat /proc/bus/pci/devices | cut -f1
0000
0001
0002
0010
0020
0030
0038
0048
0049
004a
0060
0078
0080
0090
0098
00a0
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
|-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
|-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
`-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
```

所有的 3 个设备列表都以相同顺序排列，因为 lspci 使用 /proc 文件作为它的信息源. 拿 VGA 视频控制器作一个例子，0x00a 意思是 0000:00:14.0 当划分为域(16 位)，总线(8 位)，设备(5 位)和功能(3 位).

每个外设板的硬件电路回应查询，固定在 3 个地址空间：内存位置，I/O 端口，和配置寄存器. 前 2 个地址空间由所有在同一个 PCI 总线上的设备共享(即，当你存取一个内存位置，在那个 PCI 总线上的所有的设备在同一时间都看到总线周期). 配置空间，另外的，采用地理式寻址. 配置只一次一个插槽地查询地址，因此它们从不冲突.

至于驱动，内存和 I/O 区用通常的方法，通过 inb，readb，等等来存取. 另一方面，配置传输通过调用特殊的内核函数来存取配置寄存器. 考虑到中断，每个 PCI 插槽有 4 个中断脚，并且每个设备功能可以使用它们中的一个，不必关心这些引脚如何连入 CPU. 这

样的连接是计算机平台的责任并且是在 PCI 总线之外实现的. 因为 PCI 规范要求中断线是可共享的, 即便一个处理器有有限的 IRQ 线数, 例如 x86, 可以驻有许多 PCI 接口板(每个有 4 个中断脚).

PCI 总线的 I/O 空间使用一个 32-位地址总线(产生了 4 GB 的 I/O 端口), 而内存空间可使用 32-位或者 64-位地址存取. 64-位地址在大部分近期的平台上可用. 假设地址对每个设备是唯一的, 但是软件可能错误地配置 2 个设备到同样的地址, 使得不可能存取任何一个. 但是这个问题不会产生, 除非一个驱动想玩弄不应当触动的寄存器. 好消息是每个由接口板提供的内存和 I/O 地址区可被重新映射, 通过配置交易. 那是, 在系统启动时固件初始化 PCI 硬件, 映射每个区到不同地址来避免冲突.[41]这些区当前被映射到的地址可从配置空间读出, 因此 Linux 驱动可存取它的设备而不用探测. 在读取了配置寄存器后, 驱动可安全地存取它的硬件.

PCI 配置空间为每个设备包含 256 字节(除了 PCI Express 设备, , 它每个功能有 4 KB 地配置空间), 并且配置寄存器的排布是标准化的. 配置空间的 4 个字节含有一个唯一的功能 ID, 因此驱动可标识它的设备, 通过查找那个设备的特定的 ID.[42] 总之, 每个设备板被地理式寻址来获取它的配置寄存器; 这些寄存器中的信息可接着被用来进行正常的 I/O 存取, 不必进一步的地理式寻址.

从这个描述应当清楚, PCI 接口标准对比 ISA 主要的创新是配置地址空间. 因此, 除了通常的驱动代码, 一个 PCI 驱动需要存取配置空间的能力, 为了从冒险的探测任务中解放自己.

本章的剩余部分, 我们使用词语设备来指一个设备功能, 因为在多功能板的每个功能如同一个独立的实体. 当我们引用一个设备, 我们的意思是"域号, 总线号, 设备号, 和功能号"的组合.

12.1.2. 启动时间

为见到 PCI 如何工作的, 我们从系统启动开始, 因为那是设备被配置的时候.

当一个 PCI 设备上电时, 硬件保持非激活. 换句话说, 设备只响应配置交易. 在上电时, 设备没有内存并且没有 I/O 端口被映射在计算机的地址空间; 每个其他的设备特定的特性, 例如中断报告, 也被关闭.

幸运的是, 每个 PCI 主板都装配有识别 PCI 固件, 称为 BIOS, NVRAM, 或者 PROM, 依赖平台. 这个固件提供对设备配置地址空间的存取, 通过读和写 PCI 控制器中的寄存器.

在系统启动时, 固件(或者 Linux 内核, 如果配置成这样)和每个 PCI 外设进行配置交易, 为了分配一个安全的位置给每个它提供的地址区. 在驱动存取设备的时候, 它的内存和 I/O 区已经被映射到处理器的地址空间. 驱动可改变这个缺省的分配, 但是它从不需要这样做.

如同被建议的一样, 用户可查看 PCI 设备列表和设备的配置寄存器, 通过读 /proc/bus/pcidevices 和 /proc/bus/pci/*/*. 前者是一个带有(16 进制)设备信息的文本文件, 并且后者是二进制文件来报告每个设备的每个配置寄存器的一个快照, 每个设备

一个文件. 在 sysfs 目录树中的单个的 PCI 设备目录可在 /sys/bus/pci/devices 中找到. 一个 PCI 设备目录包含许多不同的文件:

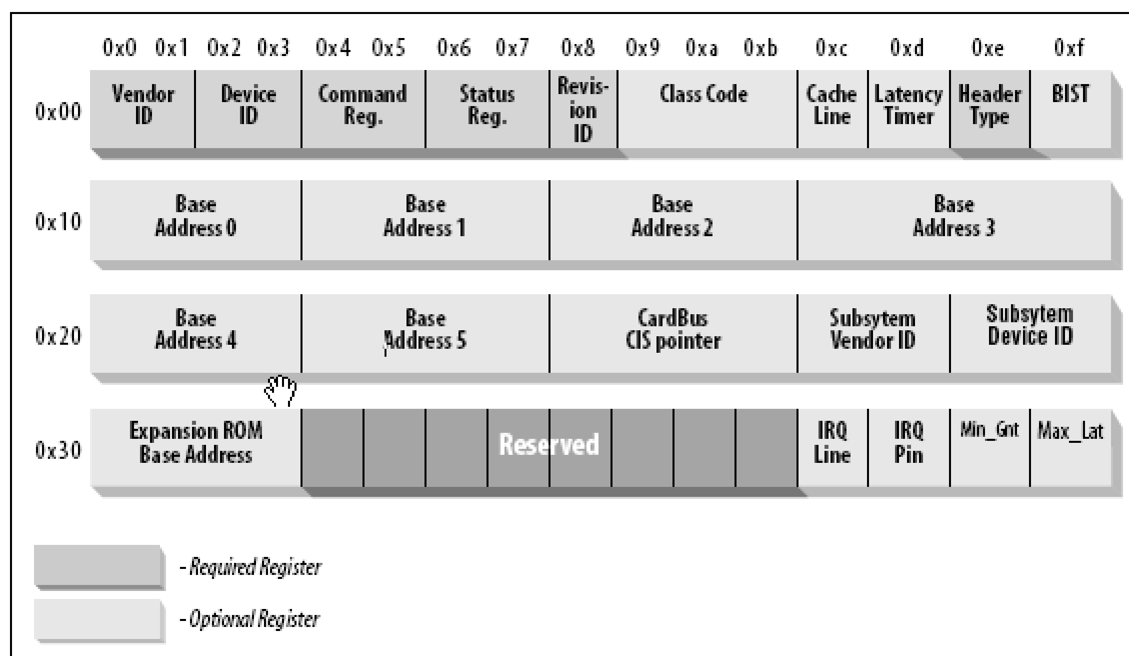
```
$ tree /sys/bus/pci/devices/0000:00:10.0
/sys/bus/pci/devices/0000:00:10.0
|-- class
|-- config
|-- detach_state
|-- device
|-- irq
|-- power
| `-- state
|-- resource
|-- subsystem_device
|-- subsystem_vendor
`-- vendor
```

文件 config 是一个二进制文件, 它允许原始的 PCI 配置信息从设备中读出(就象由 /proc/bus/pci/*/* 提供的一样). 文件 verndor, subsytem_device, subsystem_vernder, 和 class 都指的是这个 PCI 设备的特定值(所有的 PCI 设备都提供这个信息). 文件 irq 显示分配给这个 PCI 设备的当前的 IRQ, 并且文件 resource 显示这个设备分配的当前内存资源.

12.1.3. 配置寄存器和初始化

本节, 我们看 PCI 设备包含的配置寄存器. 所有的 PCI 设备都有至少一个 256-字节 地址空间. 前 64 字节是标准的, 而剩下的是依赖设备的. 图 标准 PCI 配置寄存器显示了设备独立的配置空间的排布.

图 12.2. 标准 PCI 配置寄存器



如图所示，一些 PCI 配置寄存器是要求的，一些是可选的. 每个 PCI 设备必须包含有意义的值在被要求的寄存器中，而可选寄存器的内容依赖外设的实际功能. 可选的字段不被使用，除非被要求的字段的内容指出它们是有效的. 因此，被要求的字段声称板的功能，包括其他的字段是否可用.

注意 PCI 寄存器一直是小端模式. 尽管标准被设计为独立于体系，PCI 设计者有时露出一些倾向 PC 环境. 驱动编写者应当小心处理字节序，当存取多字节配置寄存器时；在 PC 上使用的代码可能在其他平台上不工作. Linux 开发者已经处理了这个字节序问题(见下一节，"存取配置空间")，但是这个问题必须记住. 如果你曾需要转换数据从主机序到 PCI 序，或者反之，你可求助在 <asm/byteorder.h> 中定义的函数，在第 11 章介绍，知道 PCI 字节序是小端.

描述所有的配置项超出了本书的范围. 常常地，随每个设备发布的技术文档描述被支持的寄存器. 我们感兴趣的是一个驱动如何能知道它的设备以及它如何能存取设备的配置空间.

3 个或者 4 个 PCI 寄存器标识一个设备：verdorID，deviceID，和 class 是 3 个常常用到的. 每个 PCI 制造商分配正确的值给这些只读寄存器，并且驱动可使用它们来查找设备. 另外，字段 subsystem verdorID 和 subsystem deviceID 有时被供应商设置来进一步区分类似的设备.

我们看这些寄存器的细节：

vendorID

> 这个 16-位 寄存器标识一个硬件制造商. 例如，每个 Intel 设备都标有相同的供应商号，0x8086. 这样的号有一个全球的注册，由 PCI 特别利益体所维护，并且供应商必须申请有一个唯一的分配给它们的号.

deviceID

> 这是另一个 16-位 寄存器，由供应商选择；对于这个设备 ID 没有要求官方的注册. 这个 ID 常常和 供应商 ID 成对出现来组成一个唯一的 32-位 标识符给一个硬件设备. 我们使用词语"签名"来指代供应商和设备 ID 对. 一个设备驱动常常依靠签名来标识它的设备；你可在硬件手册中找到对于目标设备要寻找的值.

class

> 每个外设都属于一个类. 类寄存器是一个 16-位 值，它的高 8 位标识"基类"(或者群). 例如，"ethernet"和"token ring"是 2 个类都属于"network"群，而"serial"和"parallel"属于"communication"群. 一些驱动可支持几个类似的设备，每个都有一个不同的签名但是都属于同样的类；这些驱动可依赖类寄存器标识它们的外设，就象后面所示.

subsystem vendorID
subsystem deviceID

这些字段可用来进一步标识一个设备. 如果芯片对于本地总线是一个通用接口芯片, 它常常被用在几个完全不同的地方, 并且驱动必须标识出它在与之通话的实际设备. 子系统标志用作此目的.

使用这些不同的标识符, 一个 PCI 驱动可告知内核它支持什么类型的设备. struct pci_device_id 结构被用来定义一个驱动支持的不同类型 PCI 设备的列表. 这个结构包含不同的成员:

__u32 vendor;
__u32 device;

这些指定一个设备的 PCI 供应商和设备 ID. 如果驱动可处理任何供应商或者设备 ID, 值 PCI_ANY_ID 应当用作这些成员上.

__u32 subvendor;
__u32 subdevice;

这些指定一个设备的 PCI 子系统供应商和子系统设备 ID. 如果驱动可处理任何类型的子系统 ID, 值 PCI_ANY_ID 应当用作这些成员上.

__u32 class;
__u32 class_mask;

这 2 个值允许驱动来指定它支持一类 PCI 类设备. 不同的 PCI 设备类(一个 VAG 控制器是一个例子)在 PCI 规范里被描述. 如果一个驱动可处理任何子系统 ID, 值 PCI_ANY_ID 应当用作这些字段.

kernel_ulong_t driver_data;

这个值不用来匹配一个设备, 但是用来持有信息, PCI 驱动可用来区分不同的设备, 如果它想这样.

有 2 个帮助宏定义应当被用来初始化一个 struct pci_device_id 结构:

PCI_DEVICE(vendor, device)

这个创建一个 struct pci_device_id , 它只匹配特定的供应商和设备 ID. 这个宏设置这个结构的子供应商和子设备成员为 PCI_ANY_ID.

PCI_DEVICE_CLASS(device_class, device_class_mask)

这个创建一个 struct pci_device_id, 它匹配一个特定的 PCI 类.

一个使用这些宏来定义一个驱动支持的设备类型的例子, 在下面的内核文件中可找到:

```
drivers/usb/host/ehci-hcd.c:
static const struct pci_device_id pci_ids[] = { {
 /* handle any USB 2.0 EHCI controller */
```

```
 PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) | 0x20), ~0),
 .driver_data = (unsigned long) &ehci_driver,
 },
 { /* end: all zeroes */ }

};
drivers/i2c/busses/i2c-i810.c:

static struct pci_device_id i810_ids[] = {
 { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
 { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
 { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG) },
 { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
 { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
 { 0, },
};
```

这些例子创建一个 struct pci_device_id 结构的列表, 列表中最后一个是被设置为全零的的空结构. 这个 ID 的数组用在 struct pci_driver (下面讲述), 并且它还用来告诉用户空间这个特定的驱动支持哪个设备.

12.1.4. MODULEDEVICETABLE 宏

这个 pci_device_id 结构需要被输出到用户空间, 来允许热插拔和模块加载系统知道什么模块使用什么硬件设备. 宏 MODULE_DEVICE_TABLE 完成这个. 例如:

```
MODULE_DEVICE_TABLE(pci, i810_ids);
```

这个语句创建一个局部变量称为 __mod_pci_device_table, 它指向 struct pci_device_id 的列表. 稍后在内核建立过程中, depmod 程序在所有的模块中寻找 __mod_pci_device_table. 如果找到这个符号, 它将数据拉出模块并且添加到文件 /lib/modules/KERNEL_VERSION/modules.pcimap. 在 depmod 完成后, 所有的被内核中的模块支持的 PCI 设备被列出, 带有它们的模块名子, 在那个文件中. 当内核告知热插拔系统有新的 PCI 设备已找到, 热插拔系统使用 moudles.pcimap 文件来找到正确的驱动来加载.

12.1.5. 注册一个 PCI 驱动

为了被正确注册到内核, 所有的 PCI 驱动必须创建的主结构是 struct pci_driver 结构. 这个结构包含许多函数回调和变量, 来描述 PCI 驱动给 PCI 核心. 这里是这个结构的一个 PCI 驱动需要知道的成员:

const char *name;

> 驱动的名子. 它必须是唯一的, 在内核中所有 PCI 驱动里面. 通常被设置为和驱动模块名子相同的名子. 它显示在 sysfs 中在 /sys/bus/pci/drivers/ 下, 当驱动在内核时.

const struct pci_device_id *id_table;

指向 struct pci_device_id 表的指针，在本章后面描述它.

int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);

指向 PCI 驱动中 probe 函数的指针. 这个函数被 PCI 核心调用，当它有一个它认为这个驱动想控制的 struct pci_dev 时. 一个指向 struct pci_device_id 的指针，PCI 核心用来做这个决定的，也被传递给这个函数. 如果这个 PCI 驱动需要这个传递给它的 struct pci_dev，它应当正确初始化这个设备并且返回 0. 如果这个驱动不想拥有这个设备，或者产生一个错误，它应当返回一个负的错误值. 关于这个函数的更多的细节在本章后面.

void (*remove) (struct pci_dev *dev);

指向 PCI 核心在 struct pci_dev 被从系统中去除时调用的函数的指针，或者当 PCI 驱动被从内核中卸载时. 关于这个函数的更多的细节在本章后面.

int (*suspend) (struct pci_dev *dev, u32 state);

当 struct pci_dev 被挂起时 PCI 核心调用的函数的指针. 挂起状态在 state 变量里传递. 这个函数是可选的；一个驱动不必提供它.

int (*resume) (struct pci_dev *dev);

当 pci_dev 被恢复时 PCI 核心调用的函数的指针. 它一直被调用在调用挂起之后. 这个函数时可选的；一个驱动不必提供它.

总之，为创建一个正确的 struct pci_driver 结构，只有 4 个字段需要被初始化:

```
static struct pci_driver pci_driver = {
 .name = "pci_skel",
 .id_table = ids,
 .probe = probe,
 .remove = remove,
};
```

为注册 struct pci_driver 到 PCI 核心，用一个带有指向 struct pci_driver 的指针调用 pci_register_driver. 传统上这在 PCI 驱动的模块初始化代码中完成:

```
static int __init pci_skel_init(void)
{
 return pci_register_driver(&pci_driver);
}
```

注意，pci_register_driver 函数要么返回一个负的错误码，要么是 0 当所有都成功注册. 它不返回绑定到驱动上的设备号,或者一个错误码如果没有设备被绑定到驱动上. 这是自 2.6 发布之前的内核的一个改变，并且是因为下列的情况而来的:

- 在支持 PCI 热插拔的系统上，或者 CardBus 系统，一个 PCI 设备可在任何时间点出现或消失. 如果驱动可在设备出现前被加载是有帮助的，可以减少用来初始化一个设备的时间.
- 2.6 内核允许新 PCI ID 被动态地分配给一个驱动，在它被加载之后. 这是通过被创建在 sysfs 中的所有 PCI 驱动目录的文件 new_id 来完成的. 如果一个新设备在被使用而内核对它还不知道时，这是非常有用的. 一个用户可写 PCI ID 值到 new_id 文件，并且接着驱动绑定到新设备. 如果一个驱动不被允许加载直到一个设备在系统中出现，这个接口将不能工作.

当 PCI 驱动被卸载，struct pci_drive 需要从内核中注销. 这通过调用 pci_unregister_driver 完成. 当发生这个调用，任何当前绑定到这个驱动的 PCI 设备都被去除，并且这个 PCI 驱动的 remove 函数在 pci_unregister_driver 函数返回之前被调用.

```
static void __exit pci_skel_exit(void)
{
 pci_unregister_driver(&pci_driver);
}
```

12.1.6. 老式 PCI 探测

在老的内核版本中，函数 pci_register_driver，不是一直被 PCI 驱动使用. 相反，它们要么手工浏览系统中的 PCI 设备列表，要么它们将调用一个能够搜索一个特定 PCI 设备的函数. 驱动的浏览系统中 PCI 设备列表的能力已被从 2.6 内核中去除，为了阻止驱动破坏内核，如果它们偶尔修改 PCI 设备列表当一个设备同时被去除时.

如果发现一个特定 PCI 设备的能力真正被需要，下列的函数可用:

```
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
```

这个函数扫描当前系统中 PCI 设备的列表，并且如果输入参数匹配指定的供应商和设备 ID，它递增在 struct pci_dev 变量 found 中的引用计数，并且返回它给调用者. 这阻止了这个结构没有任何通知地消失，并且确保内核不会 oops. 在驱动用完由这个函数返回的 struct pci_dev，它必须调用函数 pci_dev_put 来正确地递减回使用计数，以允许内核清理设备如果它被去除.参数 from 用同一个签名来得到多个设备；这个参数应答指向已被找到的最后一个设备，以便搜索能够继续，而不必从列表头开始. 为找到第一个设备，from 被指定为 NULL. 如果没有找到(进一步)设备，返回 NULL.

一个如何正确使用这个函数的例子是:

```
struct pci_dev *dev;
dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
if (dev)
{
        /* Use the PCI device */
        ...
```

```
        pci_dev_put(dev);
    }
```

这个函数不能从中断上下文中被调用. 如果这样做了, 一个警告被打印到系统日志.

```
struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device,
unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
```

这个函数就象 pci_get_device 一样, 但是它允许当寻找设备时指定子系统供应商和子系统设备 ID. 这个函数不能从中断上下文调用. 如果是, 一个警告被打印到系统日志.

```
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);
```

这个函数查找系统中的 PCI 设备的列表, 在指定的 struct pci_bus 上, 一个指定的 PCI 设备的设备和功能号. 如果找到一个匹配的设备, 它的引用计数被递增并且返回指向它的一个指针. 当调用者完成存取 struct pci_dev, 它必须调用 pci_dev_put.

所有指向函数都不能从中断上下文调用. 如果是, 一个警告被打印到系统日志中.

12.1.7. 使能 PCI 设备

在 PCI 驱动的探测函数中, 在驱动可存取 PCI 设备的任何设备资源(I/O 区或者中断)之前, 驱动必须调用 pci_enable_device 函数:

```
int pci_enable_device(struct pci_dev *dev);
```

这个函数实际上使能设备. 它唤醒设备以及在某些情况下也分配它的中断线和 I/O 区. 例如, 这发生在 CardBus 设备上(它在驱动层次上已经完全和 PCI 等同了).

12.1.8. 存取配置空间

在驱动已探测到设备后, 它常常需要读或写 3 个地址空间: 内存, 端口, 和配置. 特别地, 存取配置空间对驱动是至关重要的, 因为这是唯一的找到设备被映射到内存和 I/O 空间的位置的方法.

因为微处理器无法直接存取配置空间, 计算机供应商不得不提供一个方法来完成它. 为存取配置空间, CPU 必须写和读 PCI 控制器中的寄存器, 但是确切的实现是依赖于供应商的, 并且和这个讨论无关, 因为 Linux 提供了一个标准接口来存取配置空间.

对于驱动, 配置空间可通过 8-位, 16-位, 或者 32-位数据传输来存取. 相关的函数原型定义于 <linux/pci.h>:

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
```

从由 dev 所标识出的设备的配置空间读 1 个, 2 个或者 4 个字节. where 参数是从配置空间开始的字节偏移. 从配置空间取得的值通过 val 指针返回, 并且这个函数的返回值是一个错误码. word 和 dword 函数转换刚刚读的值从小端到处理器的本地字节序, 因此你不必处理字节序.

```
int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

写 1 个, 2 个或者 4 个字节到配置空间. 象通常一样, 设备由 dev 所标识, 并且象通常一样被写的值被传递. word 和 dword 函数转换这个值到小端, 在写到外设之前.

所有的之前的函数被实现为真正调用下列函数的内联函数. 可自由使用这些函数代替上面这些, 如果这个驱动在任何特别时刻不能及时存取 struct pci_dev :

```
int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 *val);
int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16
*val);
int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32
*val);
```

就象 pci_read_function 一样, 但是 struct pci_bus * 和 devfn 变量需要来代替 struct pci_dev *.

```
int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 val);
int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16
val);
int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32
val);
```

如同 pci_write_ 函数, 但是 struct pci_bus * 和 devfn 变量需要来替代 struct pci_dev *.

使用 pci_read_ 函数寻址配置变量的最好方法是通过定义在 <linux/pci.h> 中的符号名. 例如, 下面的小函数获取一个设备的版本 ID , 通过在使用 pci_read_config_bye 时传递一个符号名.

```
static unsigned char skel_get_revision(struct pci_dev *dev)
{
 u8 revision;
 pci_read_config_byte(dev, PCI_REVISION_ID, &revision);
 return revision;
}
```

12.1.9. 存取 I/O 和内存空间

一个 PCI 设备实现直至 6 个 I/O 地址区. 每个区由要么内存要么 I/O 区组成. 大部分设备实现它们的 I/O 寄存器在内存区中, 因为通常它是一个完善的方法(如同在" I/O 端

口和 I/O 内存"一节中解释的，在第 9 章)．但是，不像正常的内存，I/O 寄存器不应当被 CPU 缓存，因为每次存取都可能有边际效果．作为内存区来实现 I/O 寄存器的 PCI 设备，通过设置一个在它的配置寄存器的"内存可预取"位来标志出这个不同．[43] 如果这个内存区被标识为可预取的，CPU 可缓存它的内容并且对它做所有类型的优化．非可预取的内存存取，另一方面，不能被优化因为每次存取可能有边际效果，就象 I/O 端口．映射它们的寄存器到一个内存地址范围的外设声明这个范围是非可预取的，而象在 PCI 板的视频内存的一些是可预取的．在本节，我们使用词语"区"来指代一个通用的 I/O 地址空间，这个空间要么是内存映射的，要么是端口映射的．

一个接口板报告它的区的大小和当前位置，使用配置寄存器- 6 个 32 位寄存器，在图 12-2 中显示的，它们的符号名是 PCI_ADDRESS_0 到 PCI_BASE_ADDRESS_5．因为 PCI 定义的 I/O 空间是 32-位空间，使用同样的配置接口给内存和 I/O 是有意义的．如果设备使用 64-位地址总线，它可以在 64-位内存空间声明各个区，使用 2 个连续的 PCI_BASE_ADDRESS 寄存器给每个区，低位在前．对一个设备可能提供 32-位 和 64-位区．

内核中，PCI 设备的 I/O 区已被集成到通用的资源管理中．由于这个原因，你不必存取配置变量来知道你的设备映射到内存或者 I/O 空间什么地方．首选的用来获得区信息的接口包括下列函数:

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
```

这个函数返回第一个地址(内存地址或者 I/O 端口号)，和 6 个 PCI I/O 区中的一个相关联的．这个区通过整数 bar (the base address register)，范围从 0-5 (包含)．

```
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
```

这个函数返回最后一个地址，I/O 区号 bar 的一部分．注意这是最后一个可用地址，不是这个区后的第一个地址．

```
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

这个函数返回和这个资源相关联的标识．

资源标识用来定义单个资源的一些特性．对于和 PCI I/O 区相关联的 PCI 资源，这个信息从基地址寄存器中抽取出来，但是可来自其他地方，对于没有和 PCI 设备关联的资源．

所有的资源标志都定义在 <linux/ioport.h>；最重要的是:

```
IORESOURCE_IO
IORESOURCE_MEM
```

如果被关联的 I/O 区存在，一个并且只有一个这样的标志被设置．

```
IORESOURCE_PREFETCH
IORESOURCE_READONLY
```

这些标志告诉是否一个内存区是可预取的并且/或者写保护的. 后一个标志对 PCI 资源从不设置.

通过使用 pci_resource_ 函数, 一个设备驱动可完全忽略底层的 PCI 寄存器, 因为系统已经使用它们来构造资源信息.

12.1.10. PCI 中断

对于中断, PCI 是容易处理的. 在 Linux 启动时, 计算机的固件已经分配一个唯一的中断号给设备, 并且驱动只需要使用它. 中断号被存储于配置寄存器 60 (PCI_INTERRUPT_LINE), 它是一个字节宽. 这允许最多 256 个中断线, 但是实际的限制依赖于使用 CPU. 驱动不必费心去检查中断号, 因为在 PCI_INTERRUPT_LINE 中找到的值保证是正确的一个.

如果设备不支持中断, 寄存器 61 (PCI_INTERRUPT_PIN) 是 0; 否则, 它是非零的值. 但是, 因为驱动知道设备是否是被中断驱动的, 它常常不需要读 PCI_INTERRUPT_PIN.

因此, 用来处理中断的 PCI 特定的代码需要读配置字节来获得保存在一个局部变量中的中断号, 如同在下面代码中显示的. 除此之外, 在第 10 章的信息适用.

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result)
{
        /* deal with error */
}
```

本节剩下的提供了额外的信息给好奇的读者, 但是对编写程序不必要.

一个 PCI 连接器有 4 个中断线, 并且外设板可使用任何一个或者多个. 每个管脚被独立连接到主板的中断控制器中, 因此中断可被共享而没有任何电路上的问题. 中断控制器接着负责映射中断线(引脚)到处理器的硬件; 这种依赖平台的操作留给控制器以便在总线自身上获得平台独立性.

位于 PCI_INTERRUPT_PIN 的只读的配置寄存器用来告知计算机实际上使用哪个管脚. 值得记住每个设备板可有多到 8 个设备; 每个设备使用一个单个中断脚并且在它的配置寄存器中报告它. 在同一个设备板上的不同设备可使用不同的中断脚或者共享同一个.

PCI_INTERRUPT_LINE 寄存器, 另一方面, 是读/写的. 当启动计算机, 固件扫描它的 PCI 设备并为每个设备设置寄存器固件中断脚是如何连接给它的 PCI 槽位. 这个值由固件分配, 因为只有固件知道主板如何连接不同的中断脚到处理器. 对于设备驱动, 但是, PCI_INTERRUPT_LINE 寄存器是只读的. 有趣的是, 近期的 Linux 内核版本在某些情况下可分配中断线, 不用依靠 BIOS.

12.1.11. 硬件抽象

我们结束 PCI 的讨论，通过快速看一下系统如何处理在市场上的多种 PCI 控制器. 这只是一个信息性的小节，打算来展示给好奇的读者，内核的面向对象分布如何向下扩展到最低层.

用来实现硬件抽象的机制是通常的包含方法的结构. 它是一个很强功能的技术，只添加最小的解引用一个指针的开销到正常的函数调用开销当中. 在 PCI 管理的情况下，唯一的硬件相关的操作是读和写配置寄存器的那些，因为在 PCI 世界中所有其他的都通过直接读和写 I/O 和内存地址空间来完成，并且那些是在 CPU 的直接控制之下.

因此，配置寄存器存取的相关的结构只包含 2 个成员:

```
struct pci_ops
{
    int (*read)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32 *val);
    int (*write)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32 val);
};
```

这个结构定义在 <linux/pci.h> 并且被 drivers/pci/pci.c 使用，这里定义了实际的公共函数.

作用于 PCI 配置空间的这 2 个函数有更大的开销，比解引用一个指针；由于代码的面向对象特性，它们使用层叠指针，但是操作中开销不是一个问题，这些操作很少被进行并且从不处于速度-关键的路径中. pci_read_config_byte(dev, where, val)的实际实现，例如，扩展为:

```
dev->bus->ops->read(bus, devfn, where, 8, val);
```

系统中各种 PCI 总线在系统启动时被探测，并且此时 struct pci_bus 项被创建并且和它们的特性所关联，包括 ops 字节.

通过"硬件操作"数据结构来实现硬件抽象在 Linux 内核中是典型的. 一个重要的例子是 struct alpha_machine_vector 数据结构. 它定义于 <asm-alpha/machvec.h> 和负责任何可能的跨不同基于 Alpha 的计算机的改变.

SBus

---

[40] 一些体系也显示 PCI 域信息在 /proc/pci 和 /proc/bus/pci 文件.

[41] 实际上，那个配置不限定在系统启动时；可热插拔的设备，例如，在启动时不可用并且相反在之后出现. 这里的要点是设备启动必须不改变 I/O 或者内存区的地址.

[42] 你将在设备自己的硬件手册里发现它的 ID. 在文件 pci.ids 中包含一个列表，这个文件是 pciutils 软件包和内核代码的一部分；它不假装是完整的，只是列出最知名的供应商和设备. 这个文件的内核版本将来不会被包含在内核系列中.

[43] 信息位于一个基地址 PCI 寄存器的低位. 这些位定义在 <linux/pci.h>.

12.2. 回顾: ISA

设计上 ISA 总线非常老了, 并且是非常地低能, 但是它仍然持有一块挺大的控制设备的市场. 如果速度不重要并且你想支持老式主板, 一个 ISA 实现要优于 PCI. 这个老标准的另外一个好处是如果你是一个电子爱好者, 你可轻易建立你自己的 ISA 设备, 显然对 PCI 是不可能的.

另一方面, ISA 的许多缺点是它紧密绑定在 PC 体系上; 这个接口总线有所有的 80286 处理器的限制并且给系统程序员带来无穷的痛苦. ISA 设计(从原始的 IBM PC 继承下来的)的另一个大问题是地理式寻址的缺乏, 它已导致许多问题和长时间的拔下-重置跳线-插上-测试 循环来添加新设备. 有趣的是要注意甚至是最老的 Aplle II 计算机都已经采用了地理式寻址, 并且它们特有无跳线扩展板.

不管它的大的缺点, ISA 仍然在几个意想不到的地方使用. 例如, 用在几个掌上电脑的 VR41xx 系列的 MIPS 处理器特有一个 ISA 兼容的扩展总线, 就象它看起来那么奇怪. 在 ISA 的这些意想不到的用法之后的理由是一些老式设备的相当低的成本, 例如基于 8390 的以太网卡, 因此一个带有 ISA 电路信号的 CPU 可轻易采用这个糟糕的, 但是便宜的 PC 设备.

12.2.1. 硬件资源

一个 ISA 设备可配备有 I/O 端口, 内存区, 和中断线.

尽管 x86 处理器支持 64 KB I/O 端口内存(即, 处理器有 16 条地址线), 一些老 PC 硬件仅解码最低的 10 位地址线. 这限制可用的地址空间为 1024 个端口, 因为任何在 1 KB 到 64 KB 范围内的地址都被只解码低地址的任何设备错当成一个低地址. 一些外设解决这个限制通过映射一个端口到低 KB 并且使用高地址线来选择不同的设备寄存器. 例如, 一个映射在 0x340 的设备可安全地使用端口 0x740, 0x840, 等等.

如果 I/O 端口的可用性被限制, 内存存取更加麻烦. 一个 ISA 设备可只使用 640KB 到 1 MB 之间的内存范围和 15 MB 和 16MB 之间的范围给 I/O 寄存器和设备控制. 640-KB 到 1-MB 范围被 PC BIOS , VAG-兼容的视频卡, 和各种其他设备使用, 给新设备留下了很少空间. 另一方面, 在 15MB 的内存, 不被 Linux 直接支持, 并且改造内核来支持它是浪费时间.

对 ISA 设备板第 3 个可用资源是中断线. 一个有限数目的中断线被连接到 ISA 总线, 并且它们由所有接口板共享. 结果是, 如果设备不被正确配置, 它们可能发现它们自己在使用同一个中断线.

尽管原始的 ISA 规范不允许在设备间共享, 大部分设备板允许这样. [44]在软件层次的共享在"中断共享"一节中描述, 在第 10 章.

12.2.2. ISA 编程

对于编程, 内核中没有特别的帮助来易于存取 ISA 设备(象对 PCI 那样有, 例如). 你可使用的唯一工具是 I/O 端口和 IRQ 线的注册, 在 10 章的"安装一个中断处理"一节.

在本书第一部分所展示的编程技术适用于 ISA 设备；驱动可探测 I/O 端口，并且中断线必须被自动探测，使用在 10 章的"自动探测 IRQ 号"一节的一个技术.

帮忙函数 isa_readb 和 它的朋友已经在"使用 I/O 内存" 9 章中简单介绍了，并且对它们没有更多要说的.

12.2.3. 即插即用规范

一些新 ISA 设备板遵循特殊的设计规范并且需要一个特别的初始化顺序，对增加接口板的简单安装和配置的扩展. 这些板的设计规范称为即插即用，由一个麻烦的规则集组成，来建立和配置无跳线的 ISA 设备. PnP 设备实现可重分配的 I/O 区；PC 的 BIOS 负责重新分配 -- 回想 PCI

简短来说，PnP 的目标是获得同样的灵活性，在 PCI 设备中有的，而不必关闭底层的电路接口(ISA 总线). 为此，这个规范定义了一套设备独立的配置寄存器和一个地理式寻址接口板的方法，尽管物理总线没有每块板子相连(地理上)--每个 ISA 信号线连接到每个可用的槽位.

地理式寻址通过分配一个小的整数，称为卡选择号(CSN)，给计算机中的每个 PnP 外设. 每个 PnP 设备特有一个唯一的系列标识符，64-位宽，这硬连线到外设板. CSN 分配使用唯一的序列号来标识 PnP 设备. 但是 CSN 可被分配只在启动时，它要求 BISO 是知道 PnP 的. 由于这个理由，老式计算机要求用户来获得并插入一个特别的配置磁盘，即便这个设备是 PnP 的.

遵循 PnP 的接口板在硬件级别上是复杂的. 它们比 PCI 板更加精细并且需要复杂的软件. 安装这些设备有困难是常有的，并且即便安装顺利，你仍然面对性能限制和 ISA 总线的受限的 I/O 空间. 最好在任何可能时安装 PCI 设备，并且享受新技术.

如果你对 PnP 配置软件感兴趣，你可浏览 drivers/net/3c509.c，它的探测函数处理 PnP 设备. 2.6 内核有许多工作在 PnP 设备支持领域，因此许多灵活的接口和之前的内核发行相比被清理了.

---

[44] 中断共享的问题是一个电子工程的问题：如果一个设备驱动信号线非激活 -- 通过给一个低阻电平 -- 中断无法被共享. 如果，另一方面，设备使用一个上拉电阻来去激活逻辑电平，共享是可能的. 现在这是正常的. 但是，仍然有潜在的丢失中断事件的危险，因为 ISA 中断是沿触发的而不是电平触发的. 沿触发中断易于在硬件中实现，但是没有使它们可安全共享.

12.3. PC/104 和 PC/104+

当前在工业世界中，2 个总线体系是非常时髦的：PC/104 和 PC/104+. 2 个在 PC-类 的单板计算机中都是标准的.

2 个标准都是印刷电路板的特殊形式，包括板互连的电子的/机械的规格. 这些总线的实际优点在它们允许电路板被垂直堆叠，使用一个在设备一边的插入并锁入的连接器.

2 个总线的电子和连接排布和 ISA(PC/104) 和 PCI(PC/104+) 是一致的, 因此软件在常用的桌面总线和它们 2 个之间, 不会注意到任何不同.

12.4. 其他的 PC 总线

PCI 和 ISA 是在 PC 世界中最常用的外设接口, 但是它们不是唯一的. 这里是对在 PC 市场上的其他总线的特性的总结.

12.4.1. MCA 总线

微通道体系(MCA)是一个 IBM 标准, 用在 PS/2 计算机和一些笔记本电脑. 在硬件级别, 微通道比 ISA 有更多特性. 它支持多主 DMA, 32-位地址和数据线, 共享中断线, 和地理式寻址来存取每块板的配置寄存器. 这样的寄存器被称为可编程选项选择(POS), 但是它们没有 PCI 寄存器的全部特点. Linux 对 微通道的支持包括输出给模块的函数.

一个设备驱动可读整数值 MCA_bus 来看是否它在一个微通道计算机上运行. 如果这个符号是一个预处理宏, 宏 MCA_bus__is_a_ 宏也被定义. 如果 MCA_bus__is_a_ 宏被取消定义, 那么 MCA_bus 是一个被输出给模块化代码的整数值. MCA_BUS 和 MCA_bus__is_a_macro 也定义在 <asm/processor.h>.

12.4.2. EISA 总线

扩展 ISA (EISA) 总线是一个对 ISA 的 32-位 扩展, 带有一个兼容的接口连接器; ISA 设备板可被插入一个 EISA 连接器. 增加的线在 ISA 接触之下被连接.

如同 PCI 和 MCA, EISA 总线被设计给无跳线的设备, 并且它有和 MCA 相同的特性: 32-位地址和数据线, 多主 DMA, 和共享中断线. EISA 设备被软件配置, 但是它们不需要任何特殊的操作系统支持. EISA 驱动已经存在于 Linux 内核给以太网驱动和 SCSI 控制器.

一个 EISA 驱动检查值 EISA_bus 来决定是否主机有一个 EISA 总线. 象 MCA_bus, EISA_bus 或者是一个宏定义或者是一个变量, 依赖是否 EISA_bus__is_a_macro 被定义. 2 个符号都定义在 <asm/processor.h>

内核对有 sysfs 和资源管理能力的设备有完整的 EISA 支持. 这位于 driver/eisa 目录.

12.4.3. VLB 总线

另一个对 ISA 的扩展是 VESA Local Bus(VLB) 接口总线, 它扩展了 ISA 连接器, 通过添加第 3 个知道长度的槽位. 一个设备可只插入这个额外的连接器(不用插入 2 个关联的 ISA 连接器), 因为 VLB 槽位从 ISA 连接器复制了所有的重要信号. 这样"独立"的 VLB 外设不使用 ISA 槽位是少见的, 因为大部分设备需要伸到后面板, 使它们的外部连接器是可用的.

VESA 总线比 EISA , MCA, 和 PCI 总线在它的能力方面更加限制, 并且正在从市场上消失. 没有特殊的内核支持位 VLB 而存在. 但是, 在 Linux 2.0 中的 Lance 以太网驱动和 IDA 磁盘驱动可处理 VLB 版本的设备.

12.5. SBus

当大部分计算机配备有 PCI 或 ISA 接口总线，大部分老式的基于 SPARC 的工作站使用 SBus 来连接它们的外设.

SBus 使一个非常先进的设计，尽管它已出现很长时间. 它意图是处理器独立的(尽管只有 SPARC 计算机使用它)并且为 I/O 外设板做了优化. 换句话说，你不能插入额外的 RAM 到 SBus 槽位(RAM 扩展板即便在 ISA 世界也已被忘记很长时间了，并且 PCI 不再支持它们). 这个优化打算来简化硬件设备和系统软件的设计，代价是主板的一些增加的复杂性.

这个总线的 I/O 偏好导致了使用虚拟地址来传送数据的外设，因此不必分配一个连续的 DMA 缓冲. 主板负责解码虚拟地址并映射它们到物理地址. 这要求连接一个 MMU(内存管理单元)到总线；负责这个任务的芯片组称为 IOMMU. 尽管比在接口总线上使用物理地址更复杂，这个设计被很大地简化，由于 SPARC 处理器一直设计为保持 MMU 内核和 CPU 内核独立(要么是物理上地，要么至少在概念上). 实际上，这个设计选择被其他的智能处理器设计所共享并且全面受益. 这个总线的另一个特性是设备板采用大块地理式寻址，因此没有必要实现一个地址解码器在每个外设或者处理地址冲突.

SBus 外设使用 Forth 语言在它们的 PROM 中来初始化它们自己. 选择 Forth 是因为解释器是轻量级的，并且因此，可轻易在任何一个计算机系统固件中实现. 另外，SBus 规范规定了驱动处理，使兼容的 I/O 设备轻易适用到系统中并且在系统启动时被识别. 这是一个大的步骤来支持多平台设备；相比我们熟悉的以 PC 为中心的 ISA 之类它是一个完全不同的世界. 但是，它不能成功，因为许多商业的原因.

尽管当前的内核版本提供了对 SBus 设备的很多全特性的支持，这个总线现在用的很少，以至于在这里它不值得详细描述. 感兴趣的读者可查看源代码 arch/sparc/kernel 和 arch/sparc/mm

12.6. NuBus 总线

另一个有趣的，但是几乎被忘记的，接口总线是 NuBus. 它被发现于老的 Mac 计算机(那些有 M68K CPU 家族的).

所有的这个总线是内存映射的(象 M68K 的所有东西)，并且设备只被地理式寻址. 这对 Apple 是好的和典型的，因为更老的 Apple II 已经有一个类似的总线布局. 不好的是几乎不可能找到 NuBus 的文档，因为 Apple 对于它的 Mac 计算机一直遵循的封锁任何东西的政策(不像之前的 Apple II，它的源码和原理图用很少的代价即可得到).

文件 drivers/nubus/nubus.c 包括几乎我们知道的关于这个总线的全部，并且读起来是有趣的；它显示了有多少难的反向过程需要开发者来做.

12.7. 外部总线

在接口总线领域的最新的一项是外部总线的整个类. 这包括 USB, 固件, 和 IEEE1284(基于并口的外部总线). 这些接口有些类似于老的非外部的技术, 例如 PCMCIA/CardBus 和甚至 SCSI.

概念上, 这些总线既不是全特性的接口总线(象 PCI 就是)也不是哑通讯通道(例如串口是). 难于划分需要来使用这些特性的软件, 因为它通常分为 2 类: 驱动, 对于硬件控制器(例如 PCI SCSI 适配器的驱动或者在"PCI 接口"一节中介绍的 PCI 控制器)以及对特殊"客户"设备的驱动(如 sd.c, 处理通用的 SCSI 磁盘和所谓的 PCI 驱动处理总线上插入的卡).

12.8. 快速参考

本节总结在本章中介绍的符号:

```
#include <linux/pci.h>
```

包含 PCI 寄存器的符号名和几个供应商和设备 ID 值的头文件.

```
struct pci_dev;
```

表示内核中一个 PCI 设备的结构.

```
struct pci_driver;
```

代表一个 PCI 驱动的结构. 所有的 PCI 驱动必须定义这个.

```
struct pci_device_id;
```

描述这个驱动支持的 PCI 设备类型的结构.

```
int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
void pci_unregister_driver(struct pci_driver *drv);
```

从内核注册或注销一个 PCI 驱动的函数.

```
struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
struct pci_dev *pci_find_device_reverse(unsigned int vendor, unsigned int device, const struct pci_dev *from);
struct pci_dev *pci_find_subsys (unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, const struct pci_dev *from);
struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);
```

在设备列表中搜寻带有一个特定签名的设备, 或者属于一个特定类的. 返回值是 NULL 如果没找到. from 用来继续一个搜索; 在你第一次调用任一个函数时它必须是 NULL, 并且它必须指向刚刚找到的设备如果你寻找更多的设备. 这些函数不推荐使用, 用 pci_get_ 变体来代替.

```
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);
```

在设备列表中搜索一个特定签名的设备，或者属于一个特定类. 返回值是 NULL 如果没找到. from 用来继续一个搜索; 在你第一次调用任一个函数时它必须是 NULL, 并且它必须指向刚刚找到的设备如果你寻找更多的设备. 返回的结构使它的引用计数递增, 并且在调用者完成它, 函数 pci_dev_put 必须被调用.

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
int pci_write_config_byte (struct pci_dev *dev, int where, u8 *val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 *val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 *val);
```

读或写 PCI 配置寄存器的函数. 尽管 Linux 内核负责字节序, 程序员必须小心字节序当从单个字节组合多字节值时. PCI 总线是小端.

```
int pci_enable_device(struct pci_dev *dev);
```

使能一个 PCI 设备.

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

处理 PCI 设备资源的函数.

第 13 章 USB 驱动

通用串行总线(USB)是一个在主机和许多外设之间的连接. 最初它被创建来替代许多慢速和不同的总线-并口, 串口, 和键盘连接--有一个单个的所有设备都可以连接的总线类型.[45] USB 已经成长超出了这些慢速连接并且现在支持几乎每种可以连接到 PC 的设备. USB 规范的最新版本增加了高速连接, 理论上到 480 MBps.

拓扑结构上, 一个 USB 子系统没有如同一个总线一样分布; 它更多是一个树, 有几个点对点连接. 这些连接是 4-线 电缆(地, 电源, 和 2 个信号线)来连接一个设备和一个集线器, 如同双绞线以太网. USB 主控制器负责询问每个 USB 设备是否它有数据发送. 由于这个拓扑关系, 一个 USB 设备在没有首先被主控制器询问时从不启动发送数据. 这个配置允许一个非常容易即插即用的系统, 这样各种设备可自动被主机配置.

在技术层面这个总线是非常简单的, 因为它是一个单主实现, 其中主机查询各种外设. 除了这个固有的限制, 这个总线有一些有趣的特性, 例如一个设备能够请求一个固定的数据传送带宽, 为了可靠地支持视频和音频 I/O. 另一个重要的特性是它只作为设备和主机之间的一个通讯通道, 对它传递的数据没有特殊的含义和结构要求.

实际上, 有一些结构, 但是它大部分精简为适应一个预先定义的类别: 例如, 一个键盘不会分配带宽, 而一些视频摄像头会.

USB 协议规范定义了一套标准, 任何特定类型的设备都可以遵循. 如果一个设备遵循这个标准, 那么给那个设备的一个特殊的驱动就不必了. 这些不同的类型称为类, 并且包含如同存储设备, 键盘, 鼠标, 游戏杆, 网络设备, 和猫. 其他不适合这些类的设备需要一个特殊的供应商-特定的驱动给这些特别的设备. 视频设备和 USB-到-串口 设备是一个好的例子, 这里没有定义好的标准, 并且需要一个驱动给每个来自不同制造商的不同的设备.

这些特性, 连同固有的设计上的热插拔能力, 使 USB 称为一个方便的, 低成本的机制来连接(和去连接)多个设备到计算机, 而不必关机, 开盒子, 并且旋开螺钉和电线.

Linux 内核支持 2 类 USB 驱动: 位于主机系统的驱动和位于设备的驱动. 给主机系统的 USB 驱动控制插入其中的 USB 设备, 从主机的观点看(一个通常的 USB 主机是一个桌面计算机). 在设备中的 USB 驱动, 控制单个设备如何作为一个 USB 设备看待主机系统. 由于术语" USB 设备驱动"是非常迷惑, USB 开发者已经创建了术语" USB 器件驱动"来描述控制一个连接到计算机的 USB 设备的驱动(记住 Linux 也运行在这些小的嵌入式的设备中). 本章详述了运行在一台桌面计算机上的 USB 系统如何工作的. USB 器件驱动此时超出了本书的范围.

如同图 USB 驱动概览所示, USB 驱动位于不同的内核子系统(块, 网络, 字符, 等等)和硬件控制器之间. USB 核心提供了一个接口给 USB 驱动用来存取和控制 USB 硬件, 而不必担心出现在系统中的不同的 USB 硬件控制器.

图 13.1. USB 驱动概览

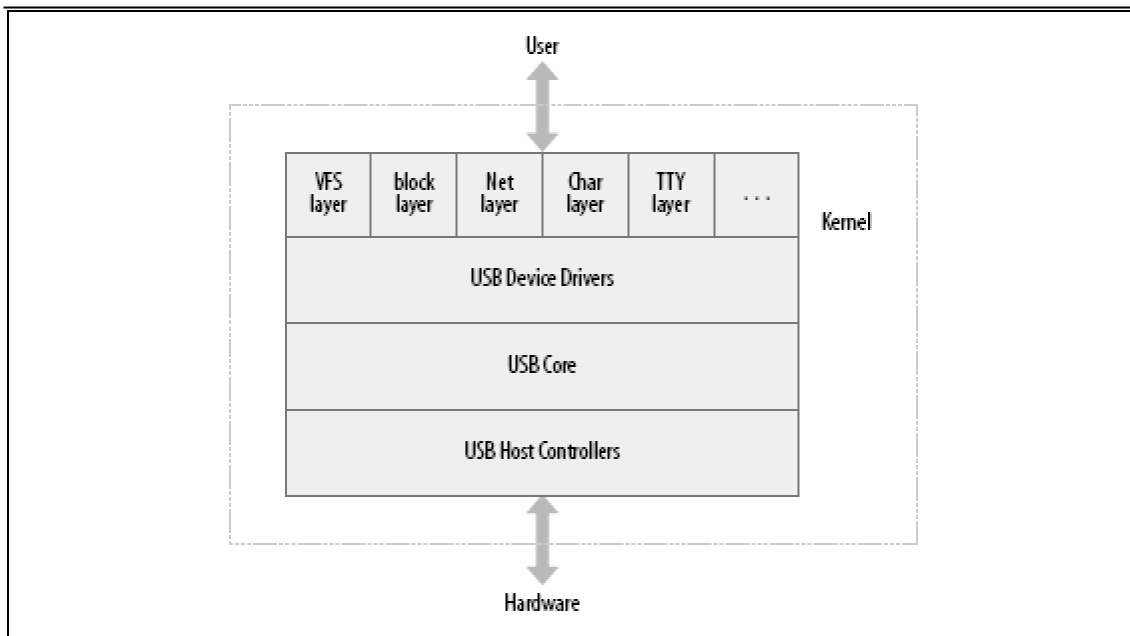
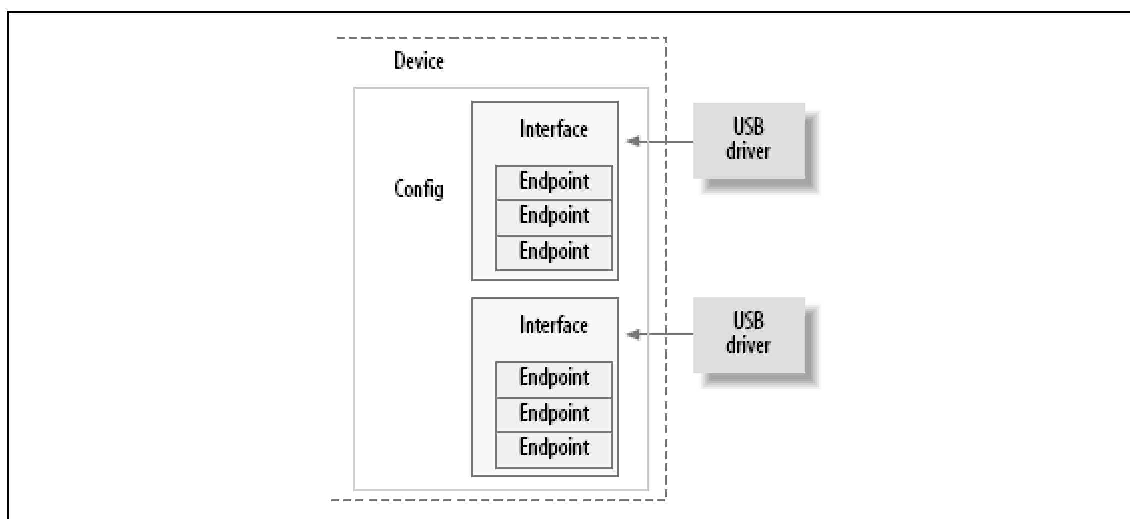


图 13.2. USB 设备概览



13.1. USB 设备基础知识

一个 USB 设备是一个非常复杂的事物，如同在官方的 USB 文档(可从 http://www.usb.org 中得到)中描述的. 幸运的是，Linux 提供了一个子系统称为 USB 核，来处理大部分复杂的工作. 这一章描述驱动和 USB 核之间的交互. 图 USB 设备概览显示了 USB 设备如何包含配置，接口，和端点，以及 USB 驱动如何绑定到 USB 接口，而不是整个 USB 设备.

13.1.1. 端点

USB 通讯的最基本形式是通过某些称为 端点 的. 一个 USB 端点只能在一个方向承载数据，或者从主机到设备(称为输出端点)或者从设备到主机(称为输入端点). 端点可看作一个单向的管道.

一个 USB 端点可是 4 种不同类型的一种，它来描述数据如何被传送:

CONTROL

控制端点被用来允许对 USB 设备的不同部分存取. 通常用作配置设备, 获取关于设备的信息, 发送命令到设备, 或者获取关于设备的状态报告. 这些端点在尺寸上常常较小. 每个 USB 设备有一个控制端点称为"端点 0", 被 USB 核用来在插入时配置设备. 这些传送由 USB 协议保证来总有足够的带宽使它到达设备.

INTERRUPT

中断端点传送小量的数据, 以固定的速率在每次 USB 主请求设备数据时. 这些端点对 USB 键盘和鼠标来说是主要的传送方法. 它们还用来传送数据到 USB 设备来控制设备, 但通常不用来传送大量数据. 这些传送由 USB 协议保证来总有足够的带宽使它到达设备.

BULK

块端点传送大量的数据. 这些端点常常比中断端点大(它们一次可持有更多的字符). 它们是普遍的, 对于需要传送不能有任何数据丢失的数据. 这些传送不被 USB 协议保证来一直使它在特定时间范围内完成. 如果总线上没有足够的空间来发送整个 BULK 报文, 它被分为多次传送到或者从设备. 这些端点普遍在打印机, 存储器, 和网络设备上.

ISOCHRONOUS

同步端点也传送大量数据, 但是这个数据常常不被保证它完成. 这些端点用在可以处理数据丢失的设备中, 并且更多依赖于保持持续的数据流. 实时数据收集, 例如音频和视频设备, 一直都使用这些端点.

控制和块端点用作异步数据传送, 无论何时驱动决定使用它们. 中断和同步端点是周期性的. 这意味着这些端点被设置来连续传送数据在固定的时间, 这使它们的带宽被 USB 核所保留.

USB 端点在内核中使用结构 struct usb_host_endpoint 来描述. 这个结构包含真实的端点信息在另一个结构中, 称为 struct usb_endpoint_descriptor. 后者包含所有的 USB-特定 数据, 以设备自身特定的准确格式. 驱动关心的这个结构的成员是:

bEndpointAddress

这是这个特定端点的 USB 地址. 还包含在这个 8-位 值的是端点的方向. 位掩码 USB_DIR_OUT 和 USB_DIR_IN 可用来和这个成员比对, 来决定给这个端点的数据是到设备还是到主机.

bmAttributes

这是端点的类型. 位掩码 USB_ENDPOINT_XFERTYPE_MASK 应当用来和这个值比对, 来决定这个端点是否是 USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, 或者是类型 USB_ENDPOINT_XFER_INT. 这些宏定义了同步, 块, 和中断端点, 相应地.

wMaxPacketSize

这是以字节计的这个端点可一次处理的最大大小. 注意驱动可能发送大量的比这个值大的数据到端点, 但是数据会被分为 wMaxPakcetSize 的块, 当真正传送到设备时. 对于高速设备, 这个成员可用来支持端点的一个高带宽模式, 通过使用几个额外位在这个值的高位部分. 关于如何完成的细节见 USB 规范.

bInterval

如果这个端点是中断类型的, 这个值是为这个端点设置的间隔, 即在请求端点的中断之间的时间. 这个值以毫秒表示.

这个结构的成员没有一个"传统" Linux 内核的命名机制. 这是因为这些成员直接对应于 USB 规范中的名子. USB 内核程序员认为使用规定的名子更重要, 以便在阅读规范时减少混乱, 不必使这些名子对 Linux 程序员看起来熟悉.

13.1.2. 接口

USB 端点被绑在接口中. USB 接口只处理一类 USB 逻辑连接, 例如一个鼠标, 一个键盘, 或者一个音频流. 一些 USB 设备有多个接口, 例如一个 USB 扬声器可能有 2 个接口: 一个 USB 键盘给按钮和一个 USB 音频流. 因为一个 USB 接口表示基本的功能, 每个 USB 驱动控制一个接口; 因此, 对扬声器的例子, Linux 需要 2 个不同的驱动给一个硬件设备.

USB 接口可能有预备的设置, 是对接口参数的不同选择. 接口的初始化的状态是第一个设置, 0 号. 预备的设置可用来以不同方式控制单独的端点, 例如来保留不同量的 USB 带宽给设备. 每个有同步端点的设备使用预备设备给同一个接口.

USB 接口在内核中使用 struct usb_interface 结构来描述. 这个结构是 USB 核传递给 USB 驱动的并且是 USB 驱动接下来负责控制的. 这个结构中的重要成员是:

struct usb_host_interface *altsetting

一个包含所有预备设置的接口结构的数组, 可被挑选给这个接口. 每个 struct usb_host_interface 包含一套端点配置, 如同由 struct usb_host_endpoint 结构所定义的. 注意这些接口结构没有特别的顺序.

unsigned num_altsetting

由 altsetting 指针指向的预备设置的数目.

struct usb_host_interface *cur_altsetting

指向数组 altsetting 的一个指针, 表示这个接口当前的激活的设置.

int minor

如果绑定到这个接口的 USB 驱动使用 USB 主编号, 这个变量包含由 USB 核心安排给接口的次编号. 这只在一次成功地调用 usb_register_dev (本章稍后描述)之后才有效.

在 struct usb_interface 结构中有其他成员, 但是 USB 驱动不需要知道它们.

13.1.3. 配置

USB 接口是自己被捆绑到配置的. 一个 USB 设备可有多个配置并且可能在它们之间转换以便改变设备的状态. 例如, 一些允许固件被下载到它们的设备包含多个配置来实现这个. 一个配置只能在一个时间点上被使能. Linux 处理多配置 USB 设备不是太好, 但是, 幸运的是, 它们很少.

linux 描述 USB 配置使用结构 struct usb_host_config 和整个 USB 设备使用结构 struct usb_device. USB 设备驱动通常不会需要读写这些结构的任何值, 因此它们在这里没有详细定义. 好奇的读者可在内核源码树的文件 include/linux/usb.h 中找到对它们的描述.

一个 USB 设备驱动通常不得不转换数据从给定的 struct usb_interface 结构到 struct usb_device 结构, USB 核心需要给很多的函数调用. 为此, 提供有函数 interface_to_usbdev. 在以后, 希望所有的当前需要一个 struct usb_device 的 USB 调用, 将被转换为采用一个 struct usb_interface 参数, 并且不会要求驱动做这个转换.

所以总结, USB 设备是非常复杂的, 并且由许多不同逻辑单元组成. 这些单元之间的关系可简单地描述如下:

- 设备通常有一个或多个配置.
- 配置常常有一个或多个接口
- 接口常常有一个或多个设置.
- 接口有零或多个端点.

---

[45] 本章的多个部分是基于内核中的给 Linux 内核 USB 代码的文档, 这些代码由内核 USB 开发者编写并且以 GPL 发布.

13.2. USB 和 sysfs

由于单个 USB 物理设备的复杂性, 设备在 sysfs 中的表示也非常复杂. 物理 USB 设备(通过 struct usb_device 表示)和单个 USB 接口(由 struct usb_interface 表示)都作为单个设备出现在 sysfs . (这是因为这 2 个结构都包含一个 struct device 结构). 例如, 对于一个简单的只包含一个 USB 接口的 USB 鼠标, 下面的内容给这个设备的 sysfs 目录树:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
```

```
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|   `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
| `-- state
|-- speed
`-- version
```

结构 usb_device 在树中被表示在:

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1

而鼠标的 USB 接口 - USB 鼠标设备驱动被绑定到的接口 - 位于目录:

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0

为帮助理解这个长设备路径的含义, 我们描述内核如何标识 USB 设备.

第一个 USB 设备是一个根集线器. 这是 USB 控制器, 常常包含在一个 PCI 设备中. 控制器的命名是由于它控制整个连接到它上面的 USB 总线. 控制器是一个 PCI 总线和 USB 总线之间的桥, 同时是总线上的第一个设备.

所有的根集线器被 USB 核心安排了一个唯一的号. 在我们的例子里, 根集线器称为 usb2, 因为它是注册到 USB 核心的第 2 个根集线器. 可包含在单个系统中在任何时间的根集线器的数目没有限制.

每个在 USB 总线上的设备采用根集线器的号作为它的名子的第一个数字. 紧跟着的是 - 字符和设备插入的端口号. 由于我们例子中的设备插在第一个端口, 一个 1 被添加到名子. 因此给主 USB 鼠标设备的名子是 2-1. 因为这个 USB 设备包含一个接口, 那使得树中的另一个设备被添加到 sysfs 路径. 到此点, USB 接口的命名方法是设备名:在我们的

例子, 是 2-1 接着一个分号和 USB 配置名, 接着一个句点和接口名. 因此对这个例子, 设备名是 2-1:1.0 因为它是第一个配置并且有接口号 0.

总结, USB sysfs 设备命名方法是:

```
root_hub-hub_port:config.interface
```

随着设备在 USB 树中进一步向下, 并且越来越多的 USB 集线器, 集线器端口号被添加到字符串中紧随着链中之前的集线器端口号. 对一个 2 层的树, 设备名看来象:

```
root_hub-hub_port-hub_port:config.interface
```

如同可在之前的 USB 设备和接口目录列表中见到的, 所有的 USB 特定信息可直接从 sysfs 获得(例如, idVendor, idProduct, 和 bMaxPower 信息). 一个文件, bConfigrationValue, 可被写入来改变激活的正被使用的 USB 配置. 这对有多个配置的设备是有用的, 当内核不能决定选择什么配置来正确操作设备. 许多 USB 猫需要有正确的配置值被写到这个文件来使正确的 USB 驱动绑定到设备.

sysfs 没暴露一个 USB 设备的所有的不同部分,因为它停止在接口水平. 任何这个设备可能包含的预备配置都没有展示, 连同关联到接口的端点的细节. 这个信息可在 usbfs 文件系统中找到, 它加载在系统的 /proc/bus/usb/ 目录. 文件 /proc/bus/usb/devices 展示了所有的在 sysfs 中暴露的信息, 连同所有的出现在系统中的 USB 设备的预备配置和端点信息. usbfs 也允许用户空间程序直接对话 USB 设备, 这已使能了许多内核驱动被移出到用户空间, 这里容易维护和调试. USB 扫描器驱动是这个的一个好例子, 由于它不再在内核中出现, 它的功能现在包含在用户空间的 SANE 库程序中.

13.3. USB 的 Urbs

linux 内核中的 USB 代码和所有的 USB 设备通讯使用称为 urb 的东西(USB request block). 这个请求块用 struct urb 结构描述并且可在 include/linux/usb.h 中找到.

一个 urb 用来发送或接受数据到或者从一个特定 USB 设备上的特定的 USB 端点, 以一种异步的方式. 它用起来非常象一个 kiocb 结构被用在文件系统异步 I/O 代码, 或者如同一个 struct skbuff 用在网络代码中. 一个 USB 设备驱动可能分配许多 urb 给一个端点或者可能重用单个 urb 给多个不同的端点, 根据驱动的需要. 设备中的每个端点都处理一个 urb 队列, 以至于多个 urb 可被发送到相同的端点, 在队列清空之前. 一个 urb 的典型生命循环如下:

- 被一个 USB 设备驱动创建.
- 安排给一个特定 USB 设备的特定端点.
- 提交给 USB 核心, 被 USB 设备驱动.
- 提交给特定设备的被 USB 核心指定的 USB 主机控制器驱动, .
- 被 USB 主机控制器处理, 它做一个 USB 传送到设备.
- 当 urb 完成, USB 主机控制器驱动通知 USB 设备驱动.

urb 也可被提交这个 urb 的驱动在任何时间取消，或者被 USB 核心如果设备被从系统中移出. urb 被动态创建并且包含一个内部引用计数，使它们在这个 urb 的最后一个用户释放它时被自动释放.

本章中描述的处理 urb 的过程是有用的，因为它允许流和其他复杂的，交叠的通讯以允许驱动来获得最高可能的数据传送速度. 但是有更少麻烦的过程可用，如果你只是想发送单独的块或者控制消息，并且不关心数据吞吐率.(见"USB 传送不用 urb"一节).

13.3.1. 结构 struct urb

struct urb 结构中和 USB 设备驱动有关的成员是:

struct usb_device *dev

指向这个 urb 要发送到的 struct usb_device 的指针. 这个变量必须被 USB 驱动初始化，在这个 urb 被发送到 USB 核心之前.

unsigned int pipe

端点消息，给这个 urb 要被发送到的特定 struct usb_device. 这个变量必须被 USB 驱动初始化，在这个 urb 被发送到 USB 核心之前.

为设置这个结构的成员，驱动使用下面的函数是适当的，依据流动的方向. 注意每个端点只可是一个类型.

unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)

指定一个控制 OUT 端点给特定的带有特定端点号的 USB 设备.

unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)

指定一个控制 IN 端点给带有特定端点号的特定 USB 设备.

unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)

指定一个块 OUT 端点给带有特定端点号的特定 USB 设备

unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)

指定一个块 IN 端点给带有特定端点号的特定 USB 设备

unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)

指定一个中断 OUT 端点给带有特定端点号的特定 USB 设备

unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)

指定一个中断 IN 端点给带有特定端点号的特定 USB 设备

unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)

指定一个同步 OUT 端点给带有特定端点号的特定 USB 设备

unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)

指定一个同步 IN 端点给带有特定端点号的特定 USB 设备

unsigned int transfer_flags

这个变量可被设置为不同位值，根据这个 USB 驱动想这个 urb 发生什么. 可用的值是:

URB_SHORT_NOT_OK

当置位，它指出任何在一个 IN 端点上可能发生的短读，应当被 USB 核心当作一个错误. 这个值只对从 USB 设备读的 urb 有用，不是写 urbs.

URB_ISO_ASAP

如果这个 urb 是同步的，这个位可被置位如果驱动想这个 urb 被调度，只要带宽允许它这样，并且在此点设置这个 urb 中的 start_frame 变量. 如果对于同步 urb 这个位没有被置位，驱动必须指定 start_frame 值并且必须能够正确恢复，如果没有在那个时刻启动. 见下面的章节关于同步 urb 更多的消息.

URB_NO_TRANSFER_DMA_MAP

应当被置位，当 urb 包含一个要被发送的 DMA 缓冲. USB 核心使用这个被 transfer_dma 变量指向的缓冲，不是被 transfer_buffer 变量指向的缓冲.

URB_NO_SETUP_DMA_MAP

象 URB_NO_TRANSFER_DMA_MAP 位，这个位用来控制有一个 DMA 缓冲已经建立的 urb. 如果它被置位，USB 核心使用这个被 setup_dma 变量而不是 setup_packet 变量指向的缓冲.

URB_ASYNC_UNLINK

如果置位，给这个 urb 的对 usb_unlink_urb 的调用几乎立刻返回，并且这个 urb 在后面被解除连接. 否则，这个函数等待直到 urb 完全被去链并且在返回前结束. 小心使用这个位，因为它可有非常难于调试的同步问题.

URB_NO_FSBR

只有 UHCI USB 主机控制器驱动使用，并且告诉它不要试图做 Front Side Bus Reclamation 逻辑. 这个位通常应当不设置，因为有 UHCI 主机控制器的机器创建了许多 CPU 负担，并且 PCI 总线被等待设置了这个位的 urb 所饱和.

URB_ZERO_PACKET

如果置位，一个块 OUT urb 通过发送不包含数据的短报文而结束，当数据对齐到一个端点报文边界. 这被一些坏掉的 USB 设备所需要(例如一些 USB 到 IR 的设备) 为了正确的工作..

URB_NO_INTERRUPT

如果置位，硬件当 urb 结束时可能不产生一个中断. 这个位应当小心使用并且只在排队多个到相同端点的 urb 时使用. USB 核心函数使用这个为了做 DMA 缓冲传送.

void *transfer_buffer

指向用在发送数据到设备(对一个 OUT urb)或者从设备中获取数据(对于一个 IN urb)的缓冲的指针. 对主机控制器为了正确存取这个缓冲，它必须被使用一个对 kmalloc 调用来创建，不是在堆栈或者静态地. 对控制端点，这个缓冲是给发送的数据阶段.

dma_addr_t transfer_dma

用来使用 DMA 传送数据到 USB 设备的缓冲.

int transfer_buffer_length

缓冲的长度，被 transfer_buffer 或者 transfer_dma 变量指向(由于只有一个可被一个 urb 使用). 如果这是 0，没有传送缓冲被 USB 核心所使用.

对于一个 OUT 端点，如果这个端点最大的大小比这个变量指定的值小，对这个 USB 设备的传送被分成更小的块为了正确的传送数据. 这种大的传送发生在连续的 USB 帧. 提交一个大块数据在一个 urb 中是非常快，并且使 USB 主机控制器去划分为更小的快，比以连续的顺序发送小缓冲.

unsigned char *setup_packet

指向给一个控制 urb 的 setup 报文的指针. 它在位于传送缓冲中的数据之前被传送. 这个变量只对控制 urb 有效.

dma_addr_t setup_dma

给控制 urb 的 setupt 报文的 DMA 缓冲. 在位于正常传送缓冲的数据之前被传送. 这个变量只对控制 urb 有效.

usb_complete_t complete

指向完成处理者函数的指针, 它被 USB 核心调用当这个 urb 被完全传送或者当 urb 发生一个错误. 在这个函数中, USB 驱动可检查这个 urb, 释放它, 或者重新提交它给另一次传送.(见"completingUrbs: 完成回调处理者", 关于完成处理者的更多细节).

usb_complete_t 类型定义如此:

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
```

void *context

指向数据点的指针, 它可被 USB 驱动设置. 它可在完成处理者中使用当 urb 被返回到驱动. 关于这个变量的细节见后续章节.

int actual_length

当这个 urb 被完成, 这个变量被设置为数据的真实长度, 或者由这个 urb (对于 OUT urb)发送或者由这个 urb(对于 IN urb)接受. 对于 IN urb, 这个必须被用来替代 transfer_buffer_length 变量, 因为接收的数据可能比整个缓冲大小小.

int status

当这个 urb 被结束, 或者开始由 USB 核心处理, 这个变量被设置为 urb 的当前状态. 一个 USB 驱动可安全存取这个变量的唯一时间是在 urb 完成处理者函数中(在"CompletingUrbs: 完成回调处理者"一节中描述). 这个限制是阻止竞争情况, 发生在这个 urb 被 USB 核心处理当中. 对于同步 urb, 在这个变量中的一个成功的值(0)只指示是否这个 urb 已被去链. 为获得在同步 urb 上的详细状态, 应当检查 iso_frame_desc 变量.

这个变量的有效值包括:

0

这个 urb 传送是成功的.

-ENOENT

这个 urb 被对 usb_kill_urb 的调用停止.

-ECONNRESET

urb 被对 usb_unlink_urb 的调用去链, 并且 transfer_flags 变量被设置为 URB_ASYNC_UNLINK.

-EINPROGRESS

这个 urb 仍然在被 USB 主机控制器处理中. 如果你的驱动曾见到这个值, 它是一个你的驱动中的 bug.

-EPROTO

这个 urb 发生下面一个错误:

- 一个 bitstuff 错误在传送中发生.
- 硬件没有及时收到响应帧.

-EILSEQ

在这个 urb 传送中有一个 CRC 不匹配.

-EPIPE

这个端点现在被停止. 如果这个包含的端点不是一个控制端点, 这个错误可被清除通过一个对函数 usb_clear_halt 的调用.

-ECOMM

在传送中数据接收快于能被写入系统内存. 这个错误值只对 IN urb.

-ENOSR

在传送中数据不能从系统内存中获取得足够快, 以便可跟上请求的 USB 数据速率. 这个错误只对 OUT urb.

-EOVERFLOW

这个 urb 发生一个"babble"错误. 一个"babble"错误发生当端点接受数据多于端点的特定最大报文大小.

-EREMOTEIO

只发生在当 URB_SHORT_NOT_OK 标志被设置在 urb 的 transfer_flags 变量, 并且意味着 urb 请求的完整数量的数据没有收到.

-ENODEV

这个 USB 设备现在从系统中消失.

-EXDEV

只对同步 urb 发生，并且意味着传送只部分完成. 为了决定传送什么，驱动必须看单独的帧状态.

-EINVAL

这个 urb 发生了非常坏的事情. USB 内核文档描述了这个值意味着什么:

ISO 疯了，如果发生这个: 退出并回家.

它也可发生，如果一个参数在 urb 结构中被不正确地设置了，或者如果在提交这个 urb 给 USB 核心的 usb_submit_urb 调用中，有一个不正确的函数参数.

-ESHUTDOWN

这个 USB 主机控制器驱动有严重的错误；它现在已被禁止，或者设备和系统去掉连接，并且这个 urb 在设备被去除后被提交. 它也可发生当这个设备的配置改变，而这个 urb 被提交给设备.

通常，错误值 -EPROTO，-EILSEQ，和 -EOVERFLOW 指示设备的硬件问题，设备固件，或者连接设备到计算机的线缆.

int start_frame

设置或返回同步传送要使用的初始帧号.

int interval

urb 被轮询的间隔. 这只对中断或者同步 urb 有效. 这个值的单位依据设备速度而不同. 对于低速和高速的设备，单位是帧，它等同于毫秒. 对于设备，单位是宏帧的设备，它等同于 1/8 微秒单位. 这个值必须被 USB 驱动设置给同步或者中断 urb，在这个 urb 被发送到 USB 核心之前.

int number_of_packets

只对同步 urb 有效，并且指定这个 urb 要处理的同步传送缓冲的编号. 这个值必须被 USB 驱动设置给同步 urb，在这个 urb 发送给 USB 核心之前.

int error_count

被 USB 核心设置，只给同步 urb 在它们完成之后. 它指定报告任何类型错误的同步传送的号码.

struct usb_iso_packet_descriptor iso_frame_desc[0]

只对同步 urb 有效. 这个变量是组成这个 urb 的一个 struct usb_iso_packet_descriptor 结构数组. 这个结构允许单个 urb 来一次定义多个同步传送. 它也用来收集每个单独传送的传送状态.

结构 usb_iso_packet_descriptor 由下列成员组成:

unsigned int offset

报文数据所在的传送缓冲中的偏移(第一个字节从 0 开始).

unsigned int length

这个报文的传送缓冲的长度.

unsigned int actual_length

接收到给这个同步报文的传送缓冲的数据长度.

unsigned int status

这个报文的单独同步传送的状态. 它可采用同样的返回值如同主 struct urb 结构的状态变量.

13.3.2. 创建和销毁 urb

struct urb 结构在驱动中必须不被静态创建, 或者在另一个结构中, 因为这可能破坏 USB 核心给 urb 使用的引用计数方法. 它必须使用对 usb_alloc_urb 函数的调用而被创建. 这个函数有这个原型:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

第一个参数, iso_packet, 是这个 urb 应当包含的同步报文的数目. 如果你不想创建一个同步 urb, 这个变量应当被设置为 0. 第 2 个参数, mem_flags, 是和传递给 kmalloc 函数调用来从内核分配内存的相同的标志类型(见"flags 参数"一节, 第 8 章, 关于这些标志的细节). 如果这个函数在分配足够内存给这个 urb 成功, 一个指向 urb 的指针被返回给调用者. 如果返回值是 NULL, 某个错误在 USB 核心中发生了, 并且驱动需要正确地清理.

在创建了一个 urb 之后, 它必须被正确初始化在它可被 USB 核心使用之前. 如何初始化不同类型 urb 见下一节

为了告诉 USB 核心驱动用完这个 urb, 驱动必须调用 usb_free_urb 函数. 这个函数只有一个参数:

```
void usb_free_urb(struct urb *urb);
```

参数是一个指向你要释放的 struct urb 的指针. 在这个函数被调用之后, urb 结构消失, 驱动不能再存取它.

13.3.2.1. 中断 urb

函数 usb_fill_int_urb 是一个帮忙函数, 来正确初始化一个 urb 来发送给 USB 设备的一个中断端点:

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
 unsigned int pipe, void *transfer_buffer,
 int buffer_length, usb_complete_t complete,
 void *context, int interval);
```

这个函数包含许多参数:

struct urb *urb

指向要被初始化的 urb 的指针.

struct usb_device *dev

这个 urb 要发送到的 USB 设备.

unsigned int pipe

这个 urb 要被发送到的 USB 设备的特定端点. 这个值被创建, 使用前面提过的 usb_sndintpipe 或者 usb_rcvintpipe 函数.

void *transfer_buffer

指向缓冲的指针, 从那里外出的数据被获取或者进入数据被接受. 注意这不能是一个静态的缓冲并且必须使用 kmalloc 调用来创建.

int buffer_length

缓冲的长度, 被 transfer_buffer 指针指向.

usb_complete_t complete

指针, 指向当这个 urb 完成时被调用的完成处理者.

void *context

指向数据块的指针, 它被添加到这个 urb 结构为以后被完成处理者函数获取.

int interval

这个 urb 应当被调度的间隔. 见之前的 struct urb 结构的描述, 来找到这个值的正确单位.

13.3.2.2. 块 urb

块 urb 被初始化非常象中断 urb. 做这个的函数是 usb_fill_bulk_urb, 它看来如此:

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
```

```
 unsigned int pipe, void *transfer_buffer,
 int buffer_length, usb_complete_t complete,
 void *context);
```

这个函数参数和 usb_fill_int_urb 函数的都相同. 但是, 没有 interval 参数因为 bulk urb 没有间隔值. 请注意这个 unsiged int pipe 变量必须被初始化用对 usb_sndbulkpipe 或者 usb_rcvbulkpipe 函数的调用.

usb_fill_int_urb 函数不设置 urb 中的 transfer_flags 变量, 因此任何对这个成员的修改不得不由这个驱动自己完成.

13.3.2.3. 控制 urb

控制 urb 被初始化几乎和 块 urb 相同的方式, 使用对函数 usb_fill_control_urb 的调用:

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
 unsigned int pipe, unsigned char *setup_packet,
 void *transfer_buffer, int buffer_length,
 usb_complete_t complete, void *context);
```

函数参数和 usb_fill_bulk_urb 函数都相同, 除了有个新参数, unsigned char *setup_packet, 它必须指向要发送给端点的 setup 报文数据. 还有, unsigned int pipe 变量必须被初始化, 使用对 usb_sndctrlpipe 或者 usb_rcvictrlpipe 函数的调用.

usb_fill_control_urb 函数不设置 transfer_flags 变量在 urb 中, 因此任何对这个成员的修改必须游驱动自己完成. 大部分驱动不使用这个函数, 因为使用在"USB 传送不用 urb"一节中介绍的同步 API 调用更简单.

13.3.2.4. 同步 urb

不幸的是, 同步 urb 没有一个象中断, 控制, 和块 urb 的初始化函数. 因此它们必须在驱动中"手动"初始化, 在它们可被提交给 USB 核心之前. 下面是一个如何正确初始化这类 urb 的例子. 它是从 konicawc.c 内核驱动中取得的, 它位于主内核源码树的 drivers/usb/media 目录.

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
```

```
 urb->iso_frame_desc[j].offset = j;
 urb->iso_frame_desc[j].length = 1;
}
```

13.3.3. 提交 urb

一旦 urb 被正确地创建,并且被 USB 驱动初始化, 它已准备好被提交给 USB 核心来发送出到 USB 设备. 这通过调用函数 usb_submit_urb 实现:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

urb 参数是一个指向 urb 的指针, 它要被发送到设备. mem_flags 参数等同于传递给 kmalloc 调用的同样的参数, 并且用来告诉 USB 核心如何及时分配任何内存缓冲在这个时间.

在 urb 被成功提交给 USB 核心之后, 应当从不试图存取 urb 结构的任何成员直到完成函数被调用.

因为函数 usb_submit_urb 可被在任何时候被调用(包括从一个中断上下文), mem_flags 变量的指定必须正确. 真正只有 3 个有效值可用, 根据何时 usb_submit_urb 被调用:

GFP_ATOMIC

这个值应当被使用无论何时下面的是真:

- 调用者处于一个 urb 完成处理者, 一个中断, 一个后半部, 一个 tasklet, 或者一个时钟回调.
- 调用者持有一个自旋锁或者读写锁. 注意如果正持有一个旗标, 这个值不必要.
- current->state 不是 TASK_RUNNING. 状态一直是 TASK_RUNNING 除非驱动已自己改变 current 状态.

GFP_NOIO

这个值应当被使用, 如果驱动在块 I/O 补丁中. 它还应当用在所有的存储类型的错误处理补丁中.

GFP_KERNEL

这应当用在所有其他的情况中, 不属于之前提到的类别.

13.3.4. 完成 urb: 完成回调处理者

如果对 usb_submit_urb 的调用成功, 传递对 urb 的控制给 USB 核心, 这个函数返回 0; 否则, 一个负错误值被返回. 如果函数成功, urb 的完成处理者(如同被完成函数指针指定的)被确切地调用一次, 当 urb 被完成. 当这个函数被调用, USB 核心完成这个 urb, 并且对它的控制现在返回给设备驱动.

只有 3 个方法, 一个 urb 可被结束并且使完成函数被调用:

- urb 被成功发送给设备，并且设备返回正确的确认．对于一个 OUT urb，数据被成功发送，对于一个 IN urb，请求的数据被成功收到．如果发生这个，urb 中的状态变量被设置为 0．
- 一些错误连续发生，当发送或者接受数据从设备中．被 urb 结构中的 status 变量中的错误值所记录．
- 这个 urb 被从 USB 核心去链．这发生在要么当驱动告知 USB 核心取消一个已提交的 urb 通过调用 usb_unlink_urb 或者 usb_kill_urb，要么当设备从系统中去除，以及一个 urb 已经被提交给它．

一个如何测试在一个 urb 完成调用中不同返回值的例子在本章稍后展示．

13.3.5. 取消 urb

为停止一个已经提交给 USB 核心的 urb，函数 usb_kill_urb 或者 usb_unlink_urb 应当被调用：

```
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

The urb parameter for both of these functions is a pointer to the urb that is to be canceled.

当函数是 usb_kill_urb，这个 urb 的生命循环就停止了．这个函数常常在设备从系统去除时被使用，在去连接回调中．

对一些驱动，应当用 usb_unlink_urb 函数来告知 USB 核心去停止 urb．这个函数在返回到调用者之前不等待这个 urb 完全停止．这对于在中断处理或者持有一个自旋锁时停止 urb 时是有用的，因为等待一个 urb 完全停止需要 USB 核心有能力使调用进程睡眠．为了正确工作这个函数要求 URB_ASYNC_UNLINK 标志值被设置在正被要求停止的 urb 中．

13.4. 编写一个 USB 驱动

编写一个 USB 设备驱动的方法类似于一个 pci 驱动：驱动注册它的驱动对象到 USB 子系统并且之后使用供应商和设备标识来告知是否它的硬件已经安装．

13.4.1. 驱动支持什么设备

struct usb_device_id 结构提供了这个驱动支持的一个不同类型 USB 设备的列表．这个列表被 USB 核心用来决定给设备哪个驱动，并且通过热插拔脚本来决定哪个驱动自动加载，当特定设备被插入系统时．

struct usb_device_id 结构定义有下面的成员：

__u16 match_flags

> 决定设备应当匹配结构中下列的哪个成员．这是一个位成员，由在 include/linux/mod_devicetable.h 文件中指定的不同的 USB_DEVICE_ID_MATCH_* 值所定义．这个成员常常从不直接设置，但是由 USB_DEVICE 类型宏来初始化．

__u16 idVendor

这个设备的 USB 供应商 ID. 这个数由 USB 论坛分配给它的成员并且不能由任何别的构成.

__u16 idProduct

这个设备的 USB 产品 ID. 所有的有分配给他们的供应商 ID 的供应商可以随意管理它们的产品 ID.

__u16 bcdDevice_lo
__u16 bcdDevice_hi

定义供应商分配的产品版本号的高低范围. bcdDevice_hi 值包括其中; 它的值是最高编号的设备号. 这 2 个值以 BCD 方式编码. 这些变量, 连同 idVendor 和 idProduct, 用来定义一个特定的设备版本.

__u8 bDeviceClass
__u8 bDeviceSubClass
__u8 bDeviceProtocol

定义类, 子类, 和设备协议, 分别地. 这些值被 USB 论坛分配并且定义在 USB 规范中. 这些值指定这个设备的行为, 包括设备上所有的接口.

__u8 bInterfaceClass
__u8 bInterfaceSubClass
__u8 bInterfaceProtocol

非常象上面的设备特定值, 这些定义了类, 子类, 和单个接口协议, 分别地. 这些值由 USB 论坛指定并且定义在 USB 规范中.

kernel_ulong_t driver_info

这个值不用来匹配, 但是它持有信息, 驱动可用来在 USB 驱动的探测回调函数区分不同的设备.

至于 PCI 设备, 有几个宏可用来初始化这个结构:

USB_DEVICE(vendor, product)

创建一个 struct usb_device_id, 可用来只匹配特定供应商和产品 ID 值. 这是非常普遍用的, 对于需要特定驱动的 USB 设备.

USB_DEVICE_VER(vendor, product, lo, hi)

创建一个 struct usb_device_id, 用来在一个版本范围中只匹配特定供应商和产品 ID 值.

USB_DEVICE_INFO(class, subclass, protocol)

创建一个 struct usb_device_id, 可用来只匹配一个特定类的 USB 设备.

USB_INTERFACE_INFO(class, subclass, protocol)

创建一个 struct usb_device_id, 可用来只匹配一个特定类的 USB 接口.

对于一个简单的 USB 设备驱动, 只控制来自一个供应商的一个单一 USB 设备, struct usb_device_id 表可定义如:

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
 { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
 { } /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

至于 PCI 驱动, MODULE_DEVICE_TABLE 宏有必要允许用户空间工具来发现这个驱动可控制什么设备. 但是对于 USB 驱动, 字符串 usb 必须是在这个宏中的第一个值.

13.4.2. 注册一个 USB 驱动

所有 USB 驱动必须创建的主要结构是 struct usb_driver. 这个结构必须被 USB 驱动填充并且包含多个函数回调和变量, 来向 USB 核心代码描述 USB 驱动:

struct module *owner

指向这个驱动的模块拥有者的指针. USB 核心使用它正确地引用计数这个 USB 驱动, 以便它不被在不合适的时刻卸载. 这个变量应当设置到 THIS_MODULE 宏.

const char *name

指向驱动名子的指针. 它必须在内核 USB 驱动中是唯一的并且通常被设置为和驱动的模块名相同. 它出现在 sysfs 中在 /sys/bus/usb/drivers/ 之下, 当驱动在内核中时.

const struct usb_device_id *id_table

指向 struct usb_device_id 表的指针, 包含这个驱动可接受的所有不同类型 USB 设备的列表. 如果这个变量没被设置, USB 驱动中的探测回调函数不会被调用. 如果你需要你的驱动给系统中每个 USB 设备一直被调用, 创建一个只设置这个 driver_info 成员的入口项:

```
static struct usb_device_id usb_ids[] = {
 {.driver_info = 42},
    {}
};
```

int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)

指向 USB 驱动中探测函数的指针. 这个函数(在"探测和去连接的细节"一节中描述)被 USB 核心调用当它认为它有一个这个驱动可处理的 struct usb_interface. 一个指向 USB 核心用来做决定的 struct usb_device_id 的指针也被传递到这个函数. 如果这个 USB 驱动主张传递给它的 struct usb_interface, 它应当正确地初始化设备并且返回 0. 如果驱动不想主张这个设备, 或者发生一个错误, 它应当返回一个负错误值.

void (*disconnect) (struct usb_interface *intf)

指向 USB 驱动的去连接函数的指针. 这个函数(在"探测和去连接的细节"一节中描述)被 USB 核心调用, 当 struct usb_interface 已被从系统中清除或者当驱动被从 USB 核心卸载.

为创建一个值 struct usb_driver 结构, 只有 5 个成员需要被初始化:

```
static struct usb_driver skel_driver = {
 .owner = THIS_MODULE,
 .name = "skeleton",
 .id_table = skel_table,
 .probe = skel_probe,
 .disconnect = skel_disconnect,
};
```

struct usb_driver 确实包含更多几个回调, 它们通常不经常用到, 并且不被要求使 USB 驱动正确工作:

int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)

指向 USB 驱动的 ioctl 函数的指针. 如果它出现, 在用户空间程序对一个关联到 USB 设备的 usbfs 文件系统设备入口, 做一个 ioctl 调用时被调用. 实际上, 只有 USB 集线器驱动使用这个 ioctl, 因为没有其他的真实需要对于任何其他 USB 驱动要使用.

int (*suspend) (struct usb_interface *intf, u32 state)

指向 USB 驱动中的悬挂函数的指针. 当设备要被 USB 核心悬挂时被调用.

int (*resume) (struct usb_interface *intf)

指向 USB 驱动中的恢复函数的指针. 当设备正被 USB 核心恢复时被调用.

为注册 struct usb_driver 到 USB 核心, 一个调用 usb_register_driver 带一个指向 struct usb_driver 的指针. 传统上在 USB 驱动的模块初始化代码做这个:

```
static int __init usb_skel_init(void)
{
        int result;
```

```
        /* register this driver with the USB subsystem */
        result = usb_register(&skel_driver);
        if (result)
                err("usb_register failed. Error number %d", result);
        return result;
}
```

当 USB 驱动被卸载, struct usb_driver 需要从内核注销. 使用对 usb_deregister_driver 的调用做这个. 当这个调用发生, 任何当前绑定到这个驱动的 USB 接口被去连接, 并且去连接函数为它们而被调用.

```
static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
```

13.4.2.1. 探测和去连接的细节

在之前章节描述的 struct usb_driver 结构中, 驱动指定 2 个 USB 核心在合适的时候调用的函数. 探测函数被调用, 当设备被安装时, USB 核心认为这个驱动应当处理; 探测函数应当进行检查传递给它的关于设备的信息, 并且决定是否驱动真正合适那个设备. 去连接函数被调用当驱动应当不再控制设备, 由于某些理由, 并且可做清理.

探测和去连接函数回调都在 USB 集线器内核线程上下文中被调用, 因此它们中睡眠是合法的. 但是, 建议如果有可能大部分工作应当在设备被用户打开时完成. 为了保持 USB 探测时间为最小. 这是因为 USB 核心处理 USB 设备的添加和去除在一个线程中, 因此任何慢设备驱动可导致 USB 设备探测时间慢下来并且用户可注意到.

在探测函数回调中, USB 驱动应当初始化任何它可能使用来管理 USB 设备的本地结构. 它还应当保存任何它需要的关于设备的信息到本地结构, 因为在此时做这些通常更容易. 作为一个例子, USB 驱动常常想为设备探测端点地址和缓冲大小是什么, 因为和设备通讯需要它们. 这里是一些例子代码, 它探测 BULK 类型的 IN 和 OUT 端点, 并且保存一些关于它们的信息在一个本地设备结构中:

```
/* set up the endpoint information */
/* use only the first bulk-in and bulk-out endpoints */
iface_desc = interface->cur_altsetting;
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i)
{

        endpoint = &iface_desc->endpoint[i].desc;
        if (!dev->bulk_in_endpointAddr &&
                        (endpoint->bEndpointAddress & USB_DIR_IN) &&
                        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)

                         == USB_ENDPOINT_XFER_BULK)) { /* we found a bulk in endpoint */
buffer_size = endpoint->wMaxPacketSize;
                dev->bulk_in_size = buffer_size;
```

```
                dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
                dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
                if (!dev->bulk_in_buffer) {
                        err("Could not allocate bulk_in_buffer");
                        goto error;
                }
        }

        if (!dev->bulk_out_endpointAddr &&
                        !(endpoint->bEndpointAddress & USB_DIR_IN) &&
                        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                         == USB_ENDPOINT_XFER_BULK)) { /* we found a bulk out endpoint */
dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
        }
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr))
{

        err("Could not find both bulk-in and bulk-out endpoints");
        goto error;
}
```

这块代码首先循环在这个接口中出现的每个端点，并且分配一个本地指针到端点结构来使它之后容易存取:

```
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
 endpoint = &iface_desc->endpoint[i].desc;
```

那么，在我们有了一个端点后，我们还没有发现一个块 IN 类型端点，我们看是否这个端点的方向是 IN. 那个可被测试通过看是否位掩码 USB_DIR_IN 被包含在 bEndpointAddress 端点变量中. 如果这是真，我们决定是否端点类型是块，通过使用 USB_ENDPOINT_XFERTYPE_MASK 位掩码首先掩去 bmAttributes 变量，并且接着检查是否它匹配值 USB_ENDPOINT_XFER_BULK:

```
if (!dev->bulk_in_endpointAddr &&
     (endpoint->bEndpointAddress & USB_DIR_IN) &&
     ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)== USB_ENDPOINT_XFER_BULK))
{
```

如果所有的这些检查都是真，驱动知道它发现了正确的端点类型，并且可保存关于端点的信息到本地结构中，它后来将需要这些信息和它通讯.

```
/* we found a bulk in endpoint */
buffer_size = endpoint->wMaxPacketSize;
dev->bulk_in_size = buffer_size;
dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer)
{
        err("Could not allocate bulk_in_buffer");
        goto error;
}
```

因为 USB 驱动需要获取在设备的生命周期后期和这个 struct usb_interface 关联的本地数据结构，函数 usb_set_intfdata 可被调用:

```
/* save our data pointer in this interface device */
usb_set_intfdata(interface, dev);
```

这个函数接受一个指向任何数据类型的指针，并且保存它到 struct usb_interface 结构为后面的存取. 为获取这个数据，函数 usb_get_intfdata 应当被调用:

```
struct usb_skel *dev;
struct usb_interface *interface;
int subminor;
int retval = 0;

subminor = iminor(inode);
interface = usb_find_interface(&skel_driver, subminor);
if (!interface)
{
        err ("%s - error, can't find device for minor %d",
             __FUNCTION__, subminor);
        retval = -ENODEV;
        goto exit;
}

dev = usb_get_intfdata(interface);
if (!dev)
{
        retval = -ENODEV;
        goto exit;
}
```

usb_get_intfdata 常常被调用，在 USB 驱动的 open 函数和在去连接函数. 感谢这 2 个函数，USB 驱动不需要保持一个静态指针数组来保存单个设备结构为系统中所有当前的设备. 对设备信息的非直接引用允许一个无限数目的设备被任何 USB 驱动支持.

如果 USB 驱动没有和另一种处理用户和设备交互的子系统(例如 input, tty, video, 等待)关联，驱动可使用 USB 主编号为了使用传统的和用户空间之间的字符驱动接口. 为此，USB 驱动必须在探测函数中调用 usb_register_dev 函数，当它想注册一个设备到 USB 核心. 确认设备和驱动处于正确的状态，来处理一个想在调用这个函数时尽快存取这个设备的用户.

```
/* we can register the device now, as it is ready */
retval = usb_register_dev(interface, &skel_class);
if (retval)
{
        /* something prevented us from registering this driver */
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
}
```

usb_register_dev 函数要求一个指向 struct usb_interface 的指针和指向 struct usb_class_driver 的指针. struct -usb_class_driver 用来定义许多不同的参数, 当注册一个次编号 USB 驱动要 USB 核心知道这些参数. 这个结构包括下列变量:.

char *name

sysfs 用来描述设备的名子. 一个前导路径名, 如果存在, 只用在 devfs 并且本书不涉及. 如果设备号需要在这个名子中, 字符 %d 应当在名子串中. 例如, 位创建 devfs 名子 usb/foo1 和 sysfs 类名 foo1, 名子串应当设置为 usb/foo%d.

struct file_operations *fops;

指向 struct file_operations 的结构的指针, 这个驱动已定义来注册为字符设备. 这个结构的更多信息见第 3 章.

mode_t mode;

给这个驱动的要被创建的 devfs 文件的模式; 否则不使用. 这个变量的典型设置是值 S_IRUSR 和 值 S_IWUSR 的结合, 它将只提供这个设备文件的拥有者读和写存取.

int minor_base;

这是给这个驱动安排的次编号的开始. 所有和这个驱动相关的设备被创建为从这个值开始的唯一的, 递增的次编号. 只有 16 个设备被允许在任何时刻和这个驱动关联, 除非 CONFIG_USB_DYNAMIC_MINORS 配置选项被打开. 如果这样, 忽略这个变量, 并且这个设备的所有的次编号被以先来先服务的方式分配. 建议打开了这个选项的系统使用一个程序例如 udev 来关联系统中的设备节点, 因为一个静态的 /dev 树不会正确工作.

当 USB 设备断开, 所有的关联到这个设备的资源应当被清除, 如果可能. 在此时, 如果 usb_register_dev 已被在探测函数中调用来分配一个 USB 设备的次编号, 函数 usb_deregister_dev 必须被调用来将次编号给回 USB 核心.

在断开函数中, 也重要的是从接口获取之前调用 usb_set_intfdata 所设置的数据. 接着设置数据指针在 struct us_interface 结构为 NULL 来阻止在不正确存取数据中的任何进一步的错误.

```
static void skel_disconnect(struct usb_interface *interface)
{
        struct usb_skel *dev;
        int minor = interface->minor;
        /* prevent skel_open() from racing skel_disconnect( ) */
        lock_kernel();

        dev = usb_get_intfdata(interface);
        usb_set_intfdata(interface, NULL);
        /* give back our minor */
        usb_deregister_dev(interface, &skel_class);
```

```
        unlock_kernel(); /* decrement our usage count */

        kref_put(&dev->kref, skel_delete);
        info("USB Skeleton #%d now disconnected", minor);
}
```

注意在之前代码片段中的调用 lock_kernel. 它获取了 bigkernel 锁, 以至于 disconnect 回调不会遇到一个竞争情况, 在使用 open 调用试图获取一个指向正确接口数据结构的指针. 因为 open 在 bigkernel 锁获取情况下被调用, 如果 disconnect 也获取同一个锁, 只有驱动的一部分可存取并且接着设置接口数据指针.

就在 disconnect 函数为一个 USB 设备被调用, 所有的当前在被传送的 urb 可被 USB 核心取消, 因此驱动不必明确为这些 urb 调用 usb_kill_urb. 如果一个驱动试图提交一个 urb 给 USB 设备, 在调用 usb_submit_urb 被断开之后, 这个任务会失败, 错误值为-EPIPE.

13.4.3. 提交和控制一个 urb

当驱动有数据发送到 USB 设备(如同在驱动的 write 函数中发生的), 一个 urb 必须被分配来传送数据到设备.

```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb)
{
        retval = -ENOMEM;
        goto error;
}
```

在 urb 被成功分配后, 一个 DMA 缓冲也应当被创建来发送数据到设备以最有效的方式, 并且被传递到驱动的数据应当被拷贝到缓冲:

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf)
{
        retval = -ENOMEM;
        goto error;
}
if (copy_from_user(buf, user_buffer, count))
{
        retval = -EFAULT;
        goto error;
}
```

应当数据被正确地从用户空间拷贝到本地缓冲, urb 在它可被提交给 USB 核心之前必须被正确初始化:

```
/* initialize the urb properly */
usb_fill_bulk_urb(urb, dev->udev,
                  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
```

```
                  buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

现在 urb 被正确分配, 数据被正确拷贝, 并且 urb 被正确初始化, 它可被提交给 USB 核心来传递给设备.

```
/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval)
{
        err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
        goto error;
}
```

在 urb 被成功传递到 USB 设备(或者在传输中发生了什么), urb 回调被 USB 核心调用. 在我们的例子中, 我们初始化 urb 来指向函数 skel_write_bulk_callback, 并且那就是被调用的函数:

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
        /* sync/async unlink faults aren't errors */
        if (urb->status &&

                        !(urb->status == -ENOENT ||
                          urb->status == -ECONNRESET ||
                          urb->status == -ESHUTDOWN)){
                dbg("%s - nonzero write bulk status received: %d",
                    __FUNCTION__, urb->status);
        }

        /* free up our allocated buffer */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                        urb->transfer_buffer, urb->transfer_dma);
}
```

回调函数做的第一件事是检查 urb 的状态来决定是否这个 urb 成功完成或没有. 错误值, -ENOENT, -ECONNRESET, 和 -ESHUTDOWN 不是真正的传送错误, 只是报告伴随成功传送的情况. (见 urb 的可能错误的列表, 在"结构 struct urb"一节中详细列出). 接着这个回调释放安排给这个 urb 传送的已分配的缓冲.

在 urb 的回调函数在运行时另一个 urb 被提交给设备是普遍的. 当流数据到设备时是有用的. 记住 urb 回调是在中断上下文运行, 因此它不应当做任何内存分配, 持有任何旗标, 或者任何可导致进程睡眠的事情. 当从回调中提交 urb, 使用 GFP_ATOMIC 标志来告知 USB 核心不要睡眠, 如果它需要分配新内存块在提交过程中.

13.5. 无 urb 的 USB 传送

有时一个 USB 驱动必须经过所有的步骤创建一个 struct urb, 初始化它, 再等待 urb 完成函数运行, 只是要发送或者接受一些简单的 USB 数据. 有 2 个函数用来提供一个简单的接口.

13.5.1. usb_bulk_msg 接口

usb_bulk_msg 创建一个 USB 块 urb 并且发送它到特定的设备, 接着在返回到调用者之前等待完成. 它定义为:

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                 void *data, int len, int *actual_length,
                 int timeout);
```

这个函数的参数是:

struct usb_device *usb_dev

发送块消息去的 USB 设备的指针

unsigned int pipe

这个块消息要发送到的 USB 设备的特定端点. 这个值被创建, 使用一个对 usb_sndbulkpipe 或者 usb_rcvbulkpipe 的调用.

void *data

如果这是一个 OUT 端点, 指向要发送到设备的数据的指针. 如果是一个 IN 端点, 这是一个在被从设备读出后数据应当被放置的地方的指针.

int len

被 data 参数指向的缓冲的长度

int *actual_length

指向函数放置真实字节数的指针, 这些字节要么被发送到设备要么从设备中获取, 根据端点方向.

int timeout

时间量, 以嘀哒计, 应当在超时前等待的. 如果这个值是 0, 函数永远等待消息完成.

如果函数成功, 返回值是 0; 否则, 一个负错误值被返回. 这错误号匹配之前在"urb 结构"一节中描述的错误号. 如果成功, actual_length 参数包含被传送或从消息中获取的字节数.

下面是一个使用这个函数调用的例子:

```
/* do a blocking bulk read to get data from the device */
retval = usb_bulk_msg(dev->udev,
                      usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
```

```
                      dev->bulk_in_buffer,
                      min(dev->bulk_in_size, count),
                      &count, HZ*10);

/* if the read was successful, copy the data to user space */
if (!retval) {
        if (copy_to_user(buffer, dev->bulk_in_buffer, count))
                retval = -EFAULT;
        else
                retval = count;
}
```

这个例子展示了一个简单的从一个 IN 端点的块读. 如果读取成功, 数据接着被拷贝到用户空间. 这个典型地是在 USB 驱动的读函数中完成.

usb_bulk_msg 函数不能被从中断上下文调用, 或者持有一个自旋锁. 还有, 这个函数不能被任何其他函数取消, 因此当使用它时小心; 确认你的驱动的去连接知道足够多来等待调用结束, 在允许它自己被从内存中卸载之前.

13.5.2. usb_control_msg 接口

usb_control_msg 函数就像 usb_bulk_msg 函数, 除了它允许一个驱动发送和结束 USB 控制信息:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8
requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);
```

这个函数的参数几乎和 usb_bulk_msg 的相同, 有几个这样的不同:

struct usb_device *dev

指向发送控制消息去的 USB 设备的指针.

unsigned int pipe

控制消息要发送到的 USB 设备的特定端点. 这个值在 usb_sndctrlpipe 或者 usb_rcvctrlpipe 函数中被创建.

__u8 request

这个控制消息的 USB 请求值.

__u8 requesttype

这个控制消息的 USB 请求类型.

__u16 value

这个控制消息的 USB 消息值.

__u16 index

这个控制消息的 USB 消息索引值.

void *data

如果是一个 OUT 端点, 是一个指向要发送到设备的数据的指针. 如果是一个 IN 端点, 是一个在被从设备读取后数据被放置的地方的指针.

__u16 size

被 data 参数指向的缓冲的大小.

int timeout

时间量, 以嘀哒计, 应当在超时前等待的. 如果这个值是 0, 这个函数将等待消息结束.

如果函数是成功的, 它返回被传送到或从这个设备的字节数. 如果它不成功, 它返回一个负错误码.

参数 request, requesttype, value, 和 index 都直接映射到 USB 规范给一个 USB 控制消息如何被定义. 对于更多的关于这些参数的有效值的信息和它们如何被使用, 见 USB 规范的第 9 章.

象 usb_bulk_msg 函数, 函数 usb_control_msg 不能被从中断上下文或者持有自旋锁中被调用. 还有, 这个函数不能被任何其他函数取消, 所以当使用它时要小心; 确认你的驱动的 disconnect 函数了解足够多, 在允许它自己被从内存卸载之前完成等待调用.

13.5.3. 使用 USB 数据函数

USB 核心中的几个帮忙函数可用来从所有的 USB 设备中存取标准信息. 这些函数不能从中断上下文或者持有自旋锁时调用.

函数 usb_get_descriptor 获取指定的 USB 描述符从特定的设备. 这个函数被定义为:

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned
char index, void *buf, int size);
```

这个函数可被一个 USB 驱动用来从 struct usb_device 结构中, 获取任何还没有在 struct usb_device 和 struct usb_interface 结构中出现的设备描述符, 例如声音描述符或者其他类的特定消息. 这个函数的参数是:

struct usb_device *usb_dev

指向应当从中获取描述符的 USB 设备的指针

unsigned char type

描述符类型. 这个类型在 USB 规范中描述, 并且是下列类型之一:

USB_DT_DEVICE USB_DT_CONFIG USB_DT_STRING USB_DT_INTERFACE USB_DT_ENDPOINT USB_DT_DEVICE_QUALIFIER USB_DT_OTHER_SPEED_CONFIG USB_DT_INTERFACE_POWER USB_DT_OTG USB_DT_DEBUG USB_DT_INTERFACE_ASSOCIATION USB_DT_CS_DEVICE USB_DT_CS_CONFIG USB_DT_CS_STRING USB_DT_CS_INTERFACE USB_DT_CS_ENDPOINT

unsigned char index

应当从设备获取的描述符的数目.

void *buf

你拷贝描述符到的缓冲的指针.

int size

由 buf 变量指向的内存的大小.

如果这个函数成功, 它返回从设备读取的字节数, 否则, 它返回由它所调用的底层函数 usb_control_msg 所返回的一个负错误值.

usb_get_descripter 调用的一项最普遍的用法是从 USB 设备获取一个字符串. 因为这个是非常普遍, 有一个帮忙函数称为 usb_get_string:

```
int usb_get_string(struct usb_device *dev, unsigned short langid, unsigned char index,
void *buf, int size);
```

如果成功, 这个函数返回设备收到的给这个字符串的字节数. 否则, 它返回一个由这个函数调用的底层函数 usb_control_msg 返回的负错误值.

如果这个函数成功, 它返回一个以 UTF-16LE 格式编码的字符串(Unicode, 16 位每字符, 小端字节序)在 buf 参数指向的缓冲中. 因为这个格式不是非常有用, 有另一个函数, 称为 usb_string, 它返回一个从一个 USB 设备读来的字符串, 并且已经转换为一个 ISO 8859-1 格式字符串. 这个字符集是一个 8 位的 UICODE 的子集, 并且是最普遍的英文和其他西欧字符串格式. 因为这是 USB 设备的字符串的典型格式, 建议 usb_string 函数来替代 usb_get_string 函数.

13.6. 快速参考

本节总结本章介绍的符号:

#include <linux/usb.h>

所有和 USB 相关的头文件. 它必须被所有的 USB 设备驱动包含.

struct usb_driver;

描述 USB 驱动的结构.

```
struct usb_device_id;
```

描述这个驱动支持的 USB 设备的结构.

```
int usb_register(struct usb_driver *d);
```

用来从 USB 核心注册和注销一个 USB 驱动的函数.

```
struct usb_device *interface_to_usbdev(struct usb_interface *intf);
```

从 struct usb_interface 获取控制 struct usb_device *.

```
struct usb_device;
```

控制完整 USB 设备的结构.

```
struct usb_interface;
```

主 USB 设备结构, 所有的 USB 驱动用来和 USB 核心通讯的.

```
void usb_set_intfdata(struct usb_interface *intf, void *data);
void *usb_get_intfdata(struct usb_interface *intf);
```

设置和获取在 struct usb_interface 中的私有数据指针部分的函数.

```
struct usb_class_driver;
```

描述 USB 驱动的一个结构, 这个驱动要使用 USB 主编号来和用户空间程序通讯.

```
int usb_register_dev(struct usb_interface *intf, struct usb_class_driver
*class_driver);
void usb_deregister_dev(struct usb_interface *intf, struct usb_class_driver
*class_driver);
```

用来注册和注销一个特定 struct usb_interface * 结构到 struct usb_class_driver 结构的函数.

```
struct urb;
```

描述一个 USB 数据传输的结构.

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
void usb_free_urb(struct urb *urb);
```

用来创建和销毁一个 struct usb urb*的函数.

```
int usb_submit_urb(struct urb *urb, int mem_flags);
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

用来启动和停止一个 USB 数据传输的函数.

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void
*transfer_buffer, int buffer_length, usb_complete_t complete, void *context, int interval);

void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void
*transfer_buffer, int buffer_length, usb_complete_t complete, void *context);

void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
unsigned char *setup_packet, void *transfer_buffer, int buffer_ length, usb_complete_t
complete, void *context);
```

用来在被提交给 USB 核心之前初始化一个 struct urb 的函数.

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, int
*actual_length, int timeout);

int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8
requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);
```

用来发送和接受 USB 数据的函数, 不必使用一个 struct urb.

第 14 章 Linux 设备模型

在 2.5 开发循环中一个声明的目标是为内核创建一个统一的设备模型. 之前的内核没有单一的数据结构, 使它们可以来获取关于系统如何整合的信息. 尽管缺乏信息, 有时事情也进行的不错. 新系统, 带有它们的更加复杂的技术并且需要支持诸如电源管理等特性, 但是, 清楚地要求需要一个通用的描述系统结构的抽象.

2.6 设备模型提供了这个抽象. 现在它用在内核来支持广泛的任务, 包括:

电源管理和系统关机

> 这些需要一个对系统的结构的理解. 例如, 一个 USB 宿主适配器不可能被关闭, 在处理所有的连接到这个适配器的设备之前. 这个设备模型使能了一个按照正确顺序的系统硬件的遍历.

与用户空间的通讯

> sysfs 虚拟文件系统的实现被紧密地捆绑进设备模型, 并且暴露它所代表的结构. 关于系统到用户空间的信息提供和改变操作参数的旋纽正越来越多地通过 sysfs 和 通过设备模型来完成.

可热插拔设备

> 计算机硬件正更多地动态变化; 外设可因用户的一时念头而进出. 在内核中使用的来处理和(特别的)与用户空间关于设备插入和拔出的通讯, 是由设备模型来管理.

设备类别

> 系统的许多部分对设备如何连接没有兴趣, 但是它们需要知道什么类型的设备可用. 设备模型包括一个机制来分配设备给类别, 它在一个更高的功能性的级别描述了这些设备, 并且允许它们从用户空间被发现.

对象生命期

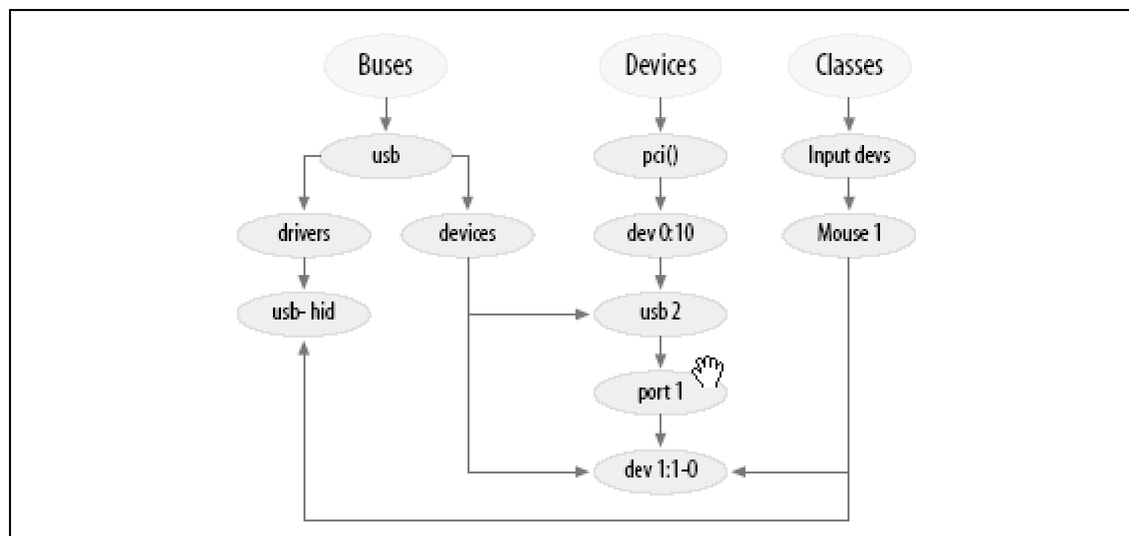
> 许多上面描述的功能, 包括热插拔支持和 sysfs, 使在内核中创建和操作对象复杂了. 设备模型的实现要求创建一套机制来处理对象生命期, 它们之间的关系, 和它们在用户空间的表示.

Linux 设备模型是一个复杂的数据结构. 例如, 考虑图设备模型的一小部分, 它展示了(用简单的形式)和 USB 鼠标关联的设备模型结构的微小片段. 图中心的下方, 我们看到核心"设备"树, 展示了鼠标如何连接到系统. "bus"树跟踪什么连接到每个总线, 而在"classes" 下的子树涉及设备提供的功能, 不管它们是如何连接的. 设备模型树即便在一个简单的系统中也包含几百个节点, 如同在图中展示的那些; 它是一个难于整个呈现的数据结构.

对大部分, Linux 设备模型代码负责所有这些方面, 而不强加自己于驱动作者之上. 它大部分位于后面; 和设备模型的直接交互通常由总线一级的逻辑和各种其他的内核子系统处理. 结果, 许多驱动作者会完全忽略设备模型, 并且信任它来照顾它自己.

有时, 但是, 理解设备模型是一个好事情. 有时设备模型从其他的层后面遛出来; 例如, 通用的 DMA 代码(我们在第 15 章遇到) 使用 struct device. 你可能想使用一些由设备模型提供的能力, 例如引用计数和由 kobjects 提供的相关特色. 通过 sysfs 和 用户空间的通讯也是一个设备模型功能; 本章解释了这个通讯如何工作.

图 14.1. 设备模型的一小部分



但是，我们开始于一个自底而上的设备模型的表述. 设备模型的复杂性使得难于从一个高层视角来理解. 我们的希望是，通过展示低层设备组件如何工作，我们可为你准备这个挑战，掌握这些组件如何用来建立更大的结构.

对大部分读者，本章可作为高级材料,不需要在第一次读完. 鼓励那些对 Linux 设备模型如何工作感兴趣的人努力向前，但是，在我们进入底层细节时.

14.1. Kobjects, Ksets 和 Subsystems

Kobject 是基础的结构，它保持设备模型在一起. 初始地它被作为一个简单的引用计数，但是它的责任已随时间增长，并且因此有了它自己的战场. struct kobject 所处理的任务和它的支持代码现在包括:

对象的引用计数

> 常常，当一个内核对象被创建，没有方法知道它会存在多长时间. 一种跟踪这种对象生命周期的方法是通过引用计数. 当没有内核代码持有对给定对象的引用，那个对象已经完成了它的有用寿命并且可以被删除.

sysfs 表示

> 在 sysfs 中出现的每个对象在它的下面都有一个 kobject，它和内核交互来创建它的可见表示.

数据结构粘和

> 设备模型是，整体来看，一个极端复杂的由多级组成的数据结构，各级之间有许多连接. kobject 实现这个结构并且保持它在一起.

热插拔事件处理

> kobject 子系统处理事件的产生，事件通知用户空间关于系统中硬件的来去.

你可能从前面的列表总结出 kobject 是一个复杂的结构. 这可能是对的. 通过一次看一部分, 但是, 是有可能理解这个结构和它如何工作的.

14.1.1. Kobject 基础

一个 kobject 有类型 struct kobject; 它在 <linux/kobject.h> 中定义. 这个文件还包含许多其他和 kobject 相关的结构的声明, 一个操作它们的函数的长列表.

14.1.1.1. 嵌入的 kobjects

在我们进入细节前, 值得花些时间理解如何使用 kobjects. 如果你回看被 kobjects 处理的函数列表, 你会看到它们都是代表其他对象进行的服务. 一个 kobject, 换句话说, 对其自己很少感兴趣; 它存在仅仅为了结合一个高级对象到设备模型.

因此, 对于内核代码它很少(甚至不知道)创建一个孤立的 kobject; 相反, kobject 被用来控制存取更大的, 特定域的对象. 为此, kobject 被嵌入到其他结构中. 如果你习惯以面向对象的术语考虑事情, kobject 可被看作一个顶级的, 抽象类, 其他的类自它而来. 一个 kobject 实现一系列功能, 这些功能对自己不是特别有用而对其他对象是好的. C 语言不允许直接表达继承, 因此其他的技术 -- 例如将一个结构嵌入另一个 -- 必须使用.

作为一个例子, 让我们回看 struct cdev, 我们在第 3 章遇到过它. 那个结构, 如同在 2.6.10 内核中发现的, 看来如此:

```
struct cdev {
 struct kobject kobj;
 struct module *owner;
 struct file_operations *ops;
 struct list_head list;
 dev_t dev;
 unsigned int count;
};
```

我们可以看出, cdev 结构有一个 kobject 嵌在里面. 如果你有一个这样的结构, 会发现它的嵌入的 kobject 只是使用 kobj 成员. 使用 kobjects 的代码有相反的问题, 但是: 如果一个 struct kobject 指针, 什么是指向包含结构的指针? 你应当避免窍门(例如假定 kobject 是在结构的开始), 并且, 相反, 使用 container_of 宏 (在第 3 章的"open 方法"一节中介绍的). 因此转换一个指向嵌在一个结构 cdev 中的一个 struct kobject 的指针 kp 的方法是:

```
struct cdev *device = container_of(kp, struct cdev, kobj);
```

程序员常常定义一个简单的宏来"后向转换" kobject 指针到包含类型.

14.1.1.2. kobject 初始化

本书已经展示了许多数据类型，带有简单的在编译或者运行时初始化机制. 一个 kobject 的初始化有些复杂，特别当使用它的所有函数时. 不管一个 kobject 如何使用，但是，必须进行几个步骤.

这些步骤的第一个是仅仅设置整个 kobject 为 0，常常使用一个对 memset 的调用. 常常这个初始化作为清零这个 kobjiect 嵌入的结构的一部分. 清零一个 kobject 失败导致非常奇怪的崩溃，进一步会掉线；这不是你想跳过的一步.

下一步是设立一些内部成员，使用对 kobject_init() 的调用：

```
void kobject_init(struct kobject *kobj);
```

在其他事情中，kobject_init 设置 kobject 的引用计数为 1. 调用 kobject_init 不够，但是. kobject 用户必须，至少，设置 kobject 的名子. 这是用在 sysfs 入口的名子. 如果你深入内核代码，你可以发现直接拷贝一个字符串到 kobject 的名子成员的代码，但是应当避免这个方法. 相反，使用：

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

这个函数采用一个 printk 风格的变量参数列表. 不管你信或不信，对这种操作实际上可能失败（ 他可能试图分配内存 ）；负责任的代码应当检查返回值并且有针对性的相应.

其他的由创建者应当设置的 kobject 成员，直接或间接，是 ktype，kset，和 parent. 我们在本章稍后到这些.

14.1.1.3. 引用计数的操作

一个 kobject 的其中一个关键函数是作为一个引用计数器，给一个它被嵌入的对象. 只要对这个对象的引用存在，这个对象（ 和支持它的代码） 必须继续存在. 来操作一个 kobject 的引用计数的低级函数是：

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

一个对 kobject_get 的成功调用递增 kobject 的 引用计数并且返回一个指向 kobject 的指针. 如果，但是，这个 kobject 已经在被销毁的过程中，这个操作失败，并且 kobject_get 返回 NULL. 这个返回值必须总是被测试，否则可能导致无法结束的令人不愉快的竞争情况.

当一个引用被释放，对 kobject_put 的调用递减引用计数，并且可能地，释放这个对象. 记住 kobject _init 设置这个引用计数为 1；因此当你创建一个 kobject，你应当确保对应地采取 kobject_put 调用，当这个初始化引用不再需要.

注意，在许多情况下，在 kobject 自身中的引用计数可能不足以阻止竞争情况. 一个 kobject 的存在(以及它的包含结构) 可能非常，例如，需要创建这个 kobject 的模块的继续存在. 在这个 kobject 仍然在被传送时不能卸载那个模块. 这是为什么我们上面看到的 cdev 结构包含一个 struct module 指针. struct cdev 的引用计数实现如下:

```
struct kobject *cdev_get(struct cdev *p)
{
 struct module *owner = p->owner;
 struct kobject *kobj;
 if (owner && !try_module_get(owner))
 return NULL;
 kobj = kobject_get(&p->kobj);
 if (!kobj)
 module_put(owner);
 return kobj;
}
```

创建一个对 cdev 结构的引用还需要创建一个对拥有它的模块的引用. 因此，cdev_get 使用 try_module_get 来试图递增这个模块的使用计数. 如果这个操作成功，kobject_get 被同样用来递增 kobject 的引用计数. 那个操作可能失败，当然，因此这个代码检查自 kobject_get 的返回值并且释放它的对模块的引用如果事情没有解决.

14.1.1.4. 释放函数和 kobject 类型

讨论中仍然缺失的一个重要事情是当一个 kobject 的引用计数到 0 时会发生什么. 创建 kobject 的代码通常不知道什么时候要发生这个情况；如果它知道，在第一位使用一个引用计数就没有意义了. 即便当引入 sysfs 时可预测的对象生命周期变得更加复杂；用户空间程序可保持一个对 kobject 的引用(通过保持一个它的关联的 sysfs 文件打开)一段任意的时间.

最后的结果是一个被 kobject 保护的结构无法在任何一个单个的，可预测的驱动生命周期中的点被释放，但是可以在必须准备在 kobject 的引用计数到 0 的任何时刻运行的代码中. 引用计数不在创建 kobject 的代码的直接控制之下. 因此这个代码必须被异步通知，无论何时对它的 kobject 的最后引用消失.

这个通知由 kobject 的一个释放函数来完成. 常常地，这个方法有一个形式如下:

```
void my_object_release(struct kobject *kobj)
{

 struct my_object *mine = container_of(kobj, struct my_object, kobj);

/* Perform any additional cleanup on this object, then... */
 kfree(mine);
}
```

要强调的重要一点是：每个 kobject 必须有一个释放函数，并且这个 kobject 必须持续(以一致的状态) 直到这个方法被调用. 如果这些限制不满足，代码就有缺陷. 当这个对象还在使用时被释放会有风险，或者在最后引用被返回后无法释放对象.

有趣的是，释放方法没有存储在 kobject 自身里面；相反，它被关联到包含 kobject 的结构类型中. 这个类型被跟踪，用一个 struct kobj_type 结构类型，常常简单地称为一个 "ktype". 这个结构看来如下:

```
struct kobj_type {
 void (*release)(struct kobject *);
 struct sysfs_ops *sysfs_ops;
 struct attribute **default_attrs;
};
```

在 struct kobj_type 中的 release 成员是，当然，一个指向这个 kobject 类型的 release 方法的指针. 我们将回到其他 2 个成员(sysfs_ops 和 default_attrs)在本章后面.

每一个 kobject 需要有一个关联的 kobj_type 结构. 易混淆地，指向这个结构的指针能在 2 个不同的地方找到. kobject 结构自身包含一个成员(称为 ktype)包含这个指针. 但是，如果这个 kobject 是一个 kset 的成员，kobj_type 指针由 kset 提供. (我们将在下一节查看 ksets.) 其间，这个宏定义:

struct kobj_type *get_ktype(struct kobject *kobj); finds the kobj_type pointer for a given kobject.

14.1.2. kobject 层次，kset，和子系统

kobject 结构常常用来连接对象到一个层级的结构中，匹配正被建模的子系统的结构. 有 2 个分开的机制对于这个连接: parent 指针和 ksets.

在结构 kobject 中的 parent 成员是一个指向其他对象的指针 -- 代表在层次中之上的下一级. 如果，例如，一个 kobject 表示一个 USB 设备，它的 parent 指针可能指示这个设备被插入的 hub.

parent 指针的主要用途是在 sysfs 层次中定位对象. 我们将看到这个如何工作，在"低级 sysfs 操作"一节中.

14.1.2.1. Ksets 对象

很多情况，一个 kset 看来象一个 kobj_type 结构的扩展；一个 kset 是一个嵌入到相同类型结构的 kobject 的集合. 但是，虽然 struct kobj_type 关注的是一个对象的类型，struct kset 被聚合和集合所关注. 这 2 个概念已被分开以至于一致类型的对象可以出现在不同的集合中.

因此，一个 kset 的主要功能是容纳；它可被当作顶层的给 kobjects 的容器类. 实际上，每个 kset 在内部容纳它自己的 kobject，并且它可以，在许多情况下，如同一个 kobject 相同的方式被对待. 值得注意的是 ksets 一直在 sysfs 中出现；一旦一个 kset 已被建立并且加入到系统，会有一个 sysfs 目录给它. kobjects 没有必要在 sysfs 中出现，但是每个是 kset 成员的 kobject 都出现在那里.

增加一个 kobject 到一个 kset 常常在一个对象创建时完成；它是一个 2 步的过程. kobject 的 kset 成员必须 ???；接着 kobject 应当被传递到:

```
int kobject_add(struct kobject *kobj);
```

如常，程序员应当小心这个函数可能失败(在这个情况下它返回一个负错误码)并且相应地反应. 有一个内核提供的方便函数:

```
extern int kobject_register(struct kobject *kobj);
```

这个函数仅仅是一个 kobject_init 和 kobject_add 的结合.

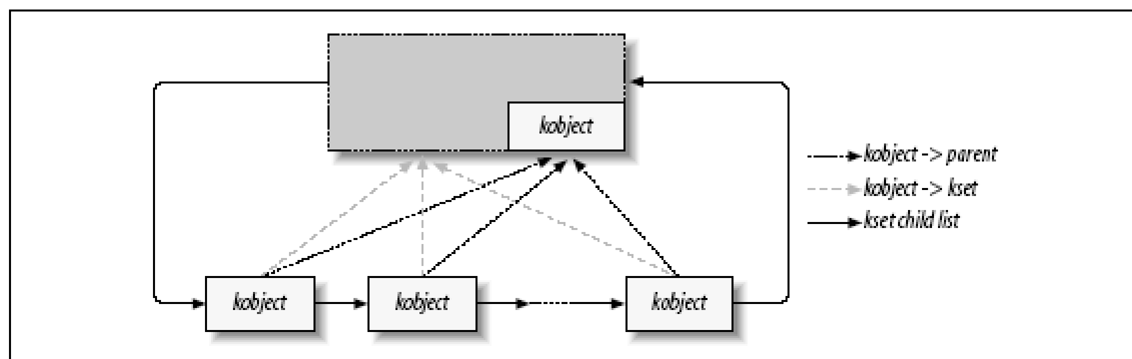
当一个 kobject 被传递给 kobject_add，它的引用计数被递增. kset 中容纳的，毕竟，是一个对这个对象的引用. 某种意义上，kobject 可能要必须从 kset 中移出来清除这个引用；完成这个使用:

```
void kobject_del(struct kobject *kobj);
```

还有一个 kobject_unregister 函数，是 kobject_del 和 kobject_put 的结合.

一个 kset 保持它的子女在一个标准的内核链表中. 在大部分情况下，被包含的 kobjects 也有指向这个 kset 的指针(或者，严格地，它的嵌入 kobject)在它们的 parent 的成员. 因此，典型地，一个 kset 和它的 kobjects 看来有些象你在图 一个简单的 kset 层次中所见. 记住:

图 14.2. 一个简单的 kset 层次



- 图表中的所有的被包含的 kobjects 实际上被嵌入在一些其他类型中，甚至可能其他的 ksets.
- 一个 kobject 的 parent 不要求是包含 kset(尽管任何其他的组织可能是奇怪的和稀少的).

14.1.2.2. ksets 之上的操作

对于初始化和设置，ksets 有一个接口非常类似于 kobjects. 下列函数存在:

```
void kset_init(struct kset *kset);
```

```
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

对大部分，这些函数只是在 kset 的嵌入对象上调用类似的 kobject_ 函数.

为管理 ksets 的引用计数，情况大概相同:

```
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
```

一个 kset 还有一个名子，存储于嵌入的 kobject. 因此，如果你有一个 kset 称为 my_set，你将设置它的名子用:

```
kobject_set_name(&my_set->kobj, "The name");
```

ksets 还有一个指针(在 ktye 成员)指向 kobject_type 结构来描述它包含的 kobject. 这个类型优先于在 kobject 自身中的 ktype 成员. 结果，在典型的应用中，在 struct kobject 中的 ktype 成员被留为 NULL，因为 kset 中的相同成员是实际使用的那个.

最后，一个 kset 包含一个子系统指针(称为 subsys). 因此是时候讨论子系统了.

14.1.2.3. 子系统

一个子系统是作为一个整体对内核一个高级部分的代表. 子系统常常(但是不是一直)出现在 sysfs 层次的顶级. 一些内核中的例子子系统包括 block_subsys(/sys/block，给块设备)，devices_subsys(/sys/devices，核心设备层次)，以及一个特殊子系统给每个内核已知的总线类型. 一个驱动作者几乎从不需要创建一个新子系统; 如果你想这样做，再仔细想想. 你可能需要什么，最后，是增加一个新类别，如同在"类别"一节中描述的.

一个子系统由一个简单结构代表:

```
struct subsystem {
 struct kset kset;
 struct rw_semaphore rwsem;
};
```

一个子系统，因此，其实只是一个对 kset 的包装，有一个旗标丢在里面.

每个 kset 必须属于一个子系统. 子系统成员关系帮助建立 kset 的位置在层次中，但是，更重要的，子系统的 rwsem 旗标用来串行化对 kset 的内部链表的存取. 这个成员关系由在 struct kset 中的 subsys 指针所表示. 因此，可以从 kset 的结构找到每个 kset 的包含子系统，但是却无法直接从子系统结构发现多个包含在子系统中的 kset.

子系统常常用一个特殊的宏声明:

```
decl_subsys(name, struct kobj_type *type, struct kset_hotplug_ops
*hotplug_ops);
```

这个宏创建一个 struct subsystem 使用一个给这个宏的名子并后缀以 _subsys 而形成的名子. 这个宏还初始化内部的 kset 使用给定的 type 和 hotplug_ops. (我们在本章后面讨论热插拔操作).

子系统有通常的建立和拆卸函数:

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

大部分这些操作只是作用在子系统的 kset 上.

14.2. 低级 sysfs 操作

kobject 是在 sysfs 虚拟文件系统之后的机制. 对每个在 sysfs 中发现的目录, 有一个 kobject 潜伏在内核某处. 每个感兴趣的 kobject 也输出一个或多个属性, 它出现在 kobject 的 sysfs 目录, 作为包含内核产生的信息的文件. 本节检查 kobject 和 sysfs 如何在低层交互.

使用 sysfs 的代码应当包含 <linux/sysfs.h>.

使一个 kobject 在 sysfs 出现仅仅是调用 kobject_add 的事情. 我们已经见到这个函数作为添加一个 kobject 到一个 kset 的方式; 在 sysfs 中创建入口也是它的工作的一部分. 有一些事情值得知道, 关于 sysfs 入口如何创建:

- kobjects 的 sysfs 入口一直为目录, 因此一个对 kobject_add 的调用导致在 sysfs 中创建一个目录. 常常地, 这个目录包含一个或多个属性; 我们稍后见到属性如何指定.
- 分配给 kobject 的名字(用 kobject_set_name) 是给 sysfs 目录使用的名字. 因此, 出现在 sysfs 层次的相同部分的 kobjects 必须有独特的名字. 分配给 kobjects 的名字也应当是合理的文件名字: 它们不能包含斜线字符, 并且空白的使用强烈不推荐.
- sysfs 入口位于对应 kobject 的 parent 指针的目录中. 如果 parent 是 NULL 当 kobject_add 被调用时, 它被设置为嵌在新 kobject 的 kset 中的 kobject; 因此, sysfs 层级常常匹配使用 kset 创建的内部层次. 如果 parent 和 kset 都是 NULL, sysfs 目录在顶级被创建, 这几乎当然不是你所要的.

使用我们至今所描述的, 我们可以使用一个 kobject 来在 sysfs 中创建一个空目录. 常常地, 你想做比这更有趣的事情, 因此是时间看属性的实现.

14.2.1. 缺省属性

当被创建时，每个 kobject 被给定一套缺省属性. 这些属性通过 kobj_type 结构来指定. 这个结构，记住，看来如此:

```
struct kobj_type {
 void (*release)(struct kobject *);
 struct sysfs_ops *sysfs_ops;
 struct attribute **default_attrs;
};
```

default_attr 成员列举了对每个这样类型的 kobject 被创建的属性，并且 sysfs_ops 提供方法来实现这些属性. 我们从 default_attrs 开始，它指向一个指向属性结构的指针数组:

```
struct attribute {
 char *name;
 struct module *owner;
 mode_t mode;
};
```

在这个结构中，name 是属性的名字(如同它出现在 kobject 的 sysfs 目录中)，owner 是一个指向模块的指针(如果有一个)，模块负责这个属性的实现，并且 mode 是应用到这个属性的保护位. mode 常常是 S_IRUGO 对于只读属性；如果这个属性是可写的，你可以扔出 S_IWUSR 来只给 root 写权限(modes 的宏定义在 <linux/stat.h> 中). default_attrs 列表中的最后一个入口必须用 0 填充.

default_attr 数组说明这些属性是什么，但是没有告诉 sysfs 如何真正实现这些属性. 这个任务落到 kobj_type->sysfs_ops 成员，它指向一个结构，定义为:

```
struct sysfs_ops {
 ssize_t (*show)(struct kobject *kobj, struct attribute *attr, char *buffer);
 ssize_t (*store)(struct kobject *kobj, struct attribute *attr, const char *buffer, size_t
size);
};
```

无论何时一个属性从用户空间读取，show 方法被用一个指向 kobject 的指针和适当的属性结构来调用. 这个方法应当将给定属性值编码进缓冲，要确定没有覆盖它(它是 PAGE_SIZE 字节)，并且返回实际的被返回数据的长度. sysfs 的惯例表明每个属性应当包含一个单个的，人可读的值；如果你有许多消息返回，你可要考虑将它分为多个属性.

同样的 show 方法用在所有的和给定 kobject 关联的属性. 传递到函数的 attr 指针可用来决定需要哪个属性. 一些 show 方法包含对属性名字的一系列测试. 其他的实现将属性结构嵌入另一个结构，来包含需要返回属性值的信息；在这种情况下，container_of 可能用在 show 方法中来获得一个指向嵌入结构的指针.

store 方法类似；它应当将存在缓冲的数据编码(size 包含数据的长度，这不能超过 PAGE_SIZE)，存储和以任何有意义的的方式响应新数据，并且返回实际编码的字节数. store 方法只在属性的许可允许写才被调用. 当编写一个 store 方法时，不要忘记你在接收来自用户空间的任意信息；你应当在采取对应动作之前非常小心地验证它. 如果到数

据不匹配期望，返回一个负的错误值，而不是可能地做一些不想要的和无法恢复的事情. 如果你的设备输出一个自销毁的属性，你应当要求一个特定的字符串写到那里来引发这个功能；一个偶然的，随机写应当只产生一个错误.

14.2.2. 非缺省属性

在许多情况中，kobject 类型的 default_attrs 成员描述所有的 kobject 会拥有的属性. 但是那不是一个设计中的限制；属性随意可以添加到和删除自 kojects. 如果你想添加一个新属性到一个 kobject 的 sysfs 目录，简单地填充一个属性结构并且传递它到：

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
```

如果所有都进行顺利，文件被使用在属性结构中给定的名字创建，并且返回值是 0；否则，返回通常的负错误码.

注意，相同的 show() 和 store() 函数被调用来实现对新属性的操作. 在你添加一个新的，非缺省属性到 kobject，你应当任何必要的步骤来确保这些函数知道如何实现这个属性.

为去除一个属性，调用：

```
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

在调用后，这个属性不再出现在 kobject 的 sysfs 入口. 要小心，但是，一个用户空间进程可能有一个打开的那个属性的文件描述符，并且在这个属性已经被去除后 show 和 store 调用仍然可能.

14.2.3. 二进制属性

sysfs 惯例调用所有属性来包含一个单个的人可读文本格式的值. 就是说，只是偶然地很少需要来创建能够处理大量二进制数据的属性. 这个需要真正地只出现在必须传递数据，不可动地，在用户空间和设备. 例如，上载固件到设备需要这个特性. 当这样一个设备在系统中遇到，一个用户程序可以被启动(通过热插拔机制)；这个程序接着传递固件代码到内核通过一个二进制 sysfs 属性，如同在"内核固件接口"一节中所示.

二进制属性使用一个 bin+attribute 结构来描述：

```
struct bin_attribute {
struct attribute attr;
size_t size;
ssize_t (*read)(struct kobject *kobj, char *buffer, loff_t pos, size_t size);
ssize_t (*write)(struct kobject *kobj, char *buffer, loff_t pos, size_t size);
};
```

这里，attr 是一个属性结构，给出名字，拥有者，和这个二进制属性的权限，并且 size 是这个二进制属性的最大大小(或者 0 ，如果没有最大值). read 和 write 方法类似于正常的字符驱动对应物；它们一次加载可被多次调用，每次调用最大一页数据. 对于

sysfs 没有办法来指示最后一个写操作，因此实现二进制属性的代码必须能够以其他方式决定数据的结束.

二进制属性必须明确创建；它们不能建立为缺省属性. 为创建一个二进制属性，调用:

```
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);
```

去除二进制属性可用:

```
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
```

14.2.4. 符号连接

sysfs 文件系统有通常的树结构，反映它代表的 kobjects 的层次组织. 但是内核中对象间的关系常常比那个更加复杂. 例如，一个 sysfs 子树 (/sys/devices)代表所有的系统已知的设备，而其他的子树(在 /sys/bus 之下)表示设备驱动. 这些树，但是，不代表驱动和它们所管理的设备间的关系. 展示这些附加关系需要额外的指针，指针在 sysfs 中通过符号连接实现.

创建一个符号连接在 sysfs 是容易的:

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
```

这个函数创建一个连接(称为 name)指向目标的 sysfs 入口作为一个 kobj 的属性. 它是一个相对连接，因此它不管 sysfs 在任何特殊的系统中安装在哪里都可用.

这个连接甚至当目标被从系统中移走也持续. 如果你在创建对其他 kobjects 的符号连接，你应当可能有一个方法知道对这个 kobjects 的改变，或者某种保证目标 kobjects 不会消失. 结果(在 sysfs 中的死的符号连接)不是特别严重，但是它们不代表最好的编程风格并且可能导致在用户空间的混乱.

去除符号连接可使用:

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

14.3. 热插拔事件产生

一个热插拔事件是一个从内核到用户空间的通知，在系统配置中有事情已经改变. 无论何时一个 kobject 被创建或销毁就产生它们. 这样事件被产生，例如，当一个数字摄像头使用一个 USB 线缆插入，当一个用户切换控制台模式，或者当一个磁盘被重新分区. 热插拔事件转变为一个对 /sbin/hotplug 的调用，它响应每个事件，通过加载驱动，创建设备节点，安装分区，或者采取任何其他的合适的动作.

我们所见的最后一个主要的 kobject 函数是这些事件的产生. 实际的事件在当一个 kobject 传递到 kobject_add 或 kobject_del 时发生. 在这个事件被传递到用户空间之前，和这个 kobject 关联的代码(或者，更特别的，它所属的 kset)有机会来添加信息给用户空间或者来完全关闭事件的产生.

14.3.1. 热插拔操作

热插拔事件的实际控制是通过一套存储于 kset_hotplug_ops 结构的方法完成.

```
struct kset_hotplug_ops {
 int (*filter)(struct kset *kset, struct kobject *kobj);
 char *(*name)(struct kset *kset, struct kobject *kobj);
 int (*hotplug)(struct kset *kset, struct kobject *kobj,
 char **envp, int num_envp, char *buffer,
 int buffer_size);
};
```

一个指向这个结构的指针在 kset 结构的 hotplug_ops 成员中. 如果一个给定的 kobject 不包含在一个 kset 中, 内核搜索整个层次(通过 parent 指针) 直到它发现一个 kobject 确实有一个 kset; 接着使用这个 kset 的热插拔操作.

filter 热插拔操作被调用无论何时内核在考虑为给定 kobject 产生一个事件. 如果 filter 返回 0, 事件没有创建. 这个方法, 因此, 给 kset 代码一个机会来决定哪个事件应当被传递给用户空间以及哪个不.

作为一个例子关于这个方法怎样被使用, 考虑块设备子系统. 至少有 3 类 kobject 用在那里, 表示磁盘, 分区, 和请求队列. 用户空间可能想对磁盘或分区的增加作出反应, 但是它正常地不关心请求队列. 因此 filter 方法允许事件产生只给代表磁盘和分区的 kobjects. 它看来如此:

```
static int block_hotplug_filter(struct kset *kset, struct kobject *kobj)
{
 struct kobj_type *ktype = get_ktype(kobj);
    return ((ktype == &ktype_block) || (ktype == &ktype_part));
}
```

这里, 一个快速的在 kobject 类型上的测试是足以决定是否这个事件应当产生或者不.

当用户空间热插拔程序被调用, 它被传递给相关子系统的 name 作为它唯一的一个参数. name 热插拔方法负责提供这个名子. 它应当返回一个简单的适合传递给用户空间的字串.

热插拔脚本的可能想知道的其他所有东东都在环境中传递. 最终的热插拔方法(hotplug) 给了一个机会来在调用这个脚本之前添加有用的环境变量. 再次, 这个方法的原型是:

```
int (*hotplug)(struct kset *kset, struct kobject *kobj,
 char **envp, int num_envp, char *buffer,
 int buffer_size);
```

如常, kset 和 kobject 描述事件产生给的对象. envp 数组是一个地方来存储额外的环境变量定义(以通常的 NAME=值 的格式); 它有 num_envp 个入口变量. 这些变量自身应当被编码入缓冲, 缓冲是 buffer_size 字节长. 如果你添加任何变量到 envp, 确信添加一个 NULL 入口在你最后的添加项后面, 这样内核知道结尾在哪里. 返回值正常应当是 0; 任何非零返回都终止热插拔事件的产生.

热插拔事件的产生(象在设备模型中大部分工作)常常是由在总线驱动级的逻辑处理.

14.4. 总线, 设备, 和驱动

至今, 我们已经看到大量低级框架和一个相对少的例子. 我们试图在本章剩下部分中补充, 随着我们进入 Linux 设备模型的更高级. 为此, 我们介绍一个新的虚拟总线, 我们称为 lddbus, [46]并且修改 scullp 驱动来 "接入" 到这个总线.

再一次, 许多驱动作者将不会需要这里涉及的材料. 这个水平的细节通常在总线级别处理, 并且很少作者需要添加一个新总线类型. 这个信息是有用的, 但是, 对任何人好奇在 PCI, USB 等层面的里面发生了什么或者谁需要在那个级别做改变.

14.4.1. 总线

一个总线是处理器和一个或多个设备之间的通道. 为设备模型的目的, 所有的设备都通过一个总线连接, 甚至当它是一个内部的虚拟的,"平台"总线. 总线可以插入另一个 - 一个 USB 控制器常常是一个 PCI 设备, 例如. 设备模型表示在总线和它们控制的设备之间的实际连接.

在 Linux 设备模型中, 一个总线由 bus_type 结构代表, 定义在 <linux/device.h>. 这个结构看来象:

```
struct bus_type {
 char *name;
 struct subsystem subsys;
 struct kset drivers;
 struct kset devices;
 int (*match)(struct device *dev, struct device_driver *drv);
 struct device *(*add)(struct device * parent, char * bus_id);
 int (*hotplug) (struct device *dev, char **envp,
 int num_envp, char *buffer, int buffer_size);
 /* Some fields omitted */
};
```

name 成员是总线的名子, 有些同 pci. 你可从这个结构中见到每个总线是它自己的子系统; 这个子系统不位于 sysfs 的顶层, 但是. 相反, 它们在总线子系统下面. 一个总线包含 2 个 ksets, 代表已知的总线的驱动和所有插入总线的设备. 所以, 有一套方法我们马上将涉及.

14.4.1.1. 总线注册

如同我们提过的, 例子源码包含一个虚拟总线实现称为 lddbus. 这个总线建立它的 bus_type 结构, 如下:

```
struct bus_type ldd_bus_type = { .name = "ldd", .match = ldd_match, .hotplug = ldd_hotplug, };
```

注意很少 bus_type 成员要求初始化；大部分由设备模型核心处理. 但是，我们确实不得不指定总线的名子，以及任何伴随它的方法.

不可避免地，一个新总线必须注册到系统，通过一个对 bus_register 的调用. lddbus 代码这样做以这样的方式：

```
ret = bus_register(&ldd_bus_type);
if (ret)
 return ret;
```

这个调用可能失败，当然，因此返回值必须一直检查. 如果它成功，新总线子系统已被添加到系统；在 sysfs 中 /sys/bus 的下面可以见到，并且可能启动添加设备.

如果有必要从系统中去除一个总线(当关联模块被去除，例如)，调用调用 bus_unregister：

```
void bus_unregister(struct bus_type *bus);
```

14.4.1.2. 总线方法

有几个给 bus_type 结构定义的方法；它们允许总线代码作为一个设备核心和单独驱动之间的中介. 在 2.6.10 内核中定义的方法是：

```
int (*match)(struct device *device, struct device_driver *driver);
```

> 这个方法被调用，大概多次，无论何时一个新设备或者驱动被添加给这个总线. 它应当返回一个非零值如果给定的设备可被给定的驱动处理. (我们马上进入设备和 device_driver 结构的细节). 这个函数必须在总线级别处理，因为那是合适的逻辑存在的地方；核心内核不能知道如何匹配每个可能总线类型的设备和驱动.

```
int (*hotplug) (struct device *device, char **envp, int num_envp, char *buffer, int buffer_size);
```

> 这个模块允许总线添加变量到环境中，在产生一个热插拔事件在用户空间之前. 参数和 kset 热插拔方法相同(在前面的 "热插拔事件产生" 一节中描述).

lddbus 驱动有一个非常简单的匹配函数，它仅仅比较驱动和设备的名子：

```
static int ldd_match(struct device *dev, struct device_driver *driver)
{
 return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
}
```

当涉及到真实硬件，match 函数常常在有设备自身提供的硬件 ID 和驱动提供的 ID 之间，做一些比较.

lddbus 热插拔方法看来象这样：

```
static int ldd_hotplug(struct device *dev, char **envp, int num_envp, char *buffer, int
buffer_size)
{
 envp[0] = buffer;
 if (snprintf(buffer, buffer_size, "LDDBUS_VERSION=%s",
 Version) >= buffer_size)
 return -ENOMEM;
 envp[1] = NULL;
 return 0;
}
```

这里, 我们加入 lddbus 源码的当前版本号, 只是以防有人好奇.

14.4.1.3. 列举设备和驱动

如果你在编写总线级别的代码, 你可能不得不对所有已经注册到你的总线的设备或驱动进行一些操作. 它可能会诱惑人直接进入 bus_type 结构中的各种结构, 但是最好使用已经提供的帮助函数.

为操作每个对总线已知的设备, 使用:

```
int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data,
int (*fn)(struct device *, void *));
```

这个函数列举总线上的每个设备, 传递关联的设备结构给 fn, 连同作为 data 来传递的值. 如果 start 是 NULL, 列举从总线的第一个设备开始; 否则列举从 start 之后的第一个设备开始. 如果 fn 返回一个非零值, 列举停止并且那个值从 bus_for_each_dev 返回.

有一个类似的函数来列举驱动:

```
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void *data, int
(*fn)(struct device_driver *, void *));
```

这个函数就像 buf_for_each_dev, 除了, 当然, 它替之作用于驱动.

应当注意, 这 2 个函数持有总线子系统的读者/写者旗标在工作期间. 因此试图一起使用这 2 个会死锁 -- 每个将试图获取同一个旗标. 修改总线的操作(例如注销设备)也将锁住. 因此, 小心使用 bus_for_each 函数.

14.4.1.4. 总线属性

几乎 Linux 驱动模型中的每一层都提供一个添加属性的接口, 并且总线层不例外. bus_attribute 类型定义在 <linux/device.h> 如下:

```
struct bus_attribute {
 struct attribute attr;
 ssize_t (*show)(struct bus_type *bus, char *buf);
```

```
 ssize_t (*store)(struct bus_type *bus, const char *buf,
 size_t count);
};
```

我们已经见到 struct attribute 在 "缺省属性" 一节. bus_attribute 类型也包含 2 个方法来显示和设置属性值. 大部分在 kobject 之上的设备模型层以这种方式工作.

已经提供了一个方便的宏为在编译时间创建和初始化 bus_attribute 结构:

```
BUS_ATTR(name, mode, show, store);
```

这个宏声明一个结构, 产生它的名子通过前缀字符串 bus_attr_ 到给定的名子.

任何属于一个总线的属性应当明确使用 bus_create_file 来创建:

```
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
```

属性也可被去除, 使用:

```
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

lddbus 驱动创建一个简单属性文件, 再次, 包含源码版本号. show 方法和 bus_attribute 结构设置如下:

```
static ssize_t show_bus_version(struct bus_type *bus, char *buf)
{
 return snprintf(buf, PAGE_SIZE, "%s\n", Version);
}

static BUS_ATTR(version, S_IRUGO, show_bus_version, NULL);
```

创建属性文件在模块加载时间完成:

```
if (bus_create_file(&ldd_bus_type, &bus_attr_version))
 printk(KERN_NOTICE "Unable to create version attribute\n");
```

这个调用创建一个属性文件(/sys/busldd/version) 包含 lddbus 代码的版本号.

14.4.2. 设备

在最低层, Linux 系统中的每个设备由一个 struct device 代表:

```
struct device { struct device *parent; struct kobject kobj; char bus_id[BUS_ID_SIZE];
struct bus_type *bus; struct device_driver *driver; void *driver_data; void
(*release)(struct device *dev); /* Several fields omitted */ };
```

有许多其他的 struct device 成员只对设备核心代码感兴趣. 但是, 这些成员值得了解:

struct device *parent

设备的 "parent" 设备 -- 它所附着到的设备. 在大部分情况, 一个父设备是某种总线或者主控制器. 如果 parent 是 NULL, 设备是一个顶层设备, 这常常不是你所要的.

struct kobject kobj;

代表这个设备并且连接它到层次中的 kobject. 注意, 作为一个通用的规则, device->kobj->parent 等同于 device->parent->kobj.

char bus_id[BUS_ID_SIZE];

唯一确定这个总线上的设备的字符串. PCI 设备, 例如, 使用标准的 PCI ID 格式, 包含域, 总线, 设备, 和功能号.

struct bus_type *bus;

确定设备位于哪种总线.

struct device_driver *driver;

管理这个设备的驱动; 我们查看 struct device_driver 在下一节.

void *driver_data;

一个可能被设备驱动使用的私有数据成员.

void (*release)(struct device *dev);

当对这个设备的最后引用被去除时调用的方法; 它从被嵌入的 kobject 的 release 方法被调用. 注册到核心的所有的设备结构必须有一个 release 方法, 否则内核打印出慌乱的抱怨.

最少, parent, bus_id, bus, 和 release 成员必须在设备结构被注册前设置.

14.4.2.1. 设备注册

通常的注册和注销函数在:

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

我们已经见到 lddbus 代码如何注册它的总线类型. 但是, 一个实际的总线是一个设备并且必须单独注册. 为简单起见, lddbus 模块只支持一个单个虚拟总线, 因此这个驱动在编译时建立它的设备:

```
static void ldd_bus_release(struct device *dev)
{
 printk(KERN_DEBUG "lddbus release\n");
```

```
}

struct device ldd_bus = {
 .bus_id = "ldd0",
 .release = ldd_bus_release

};
```

这是顶级总线，因此 parent 和 bus 成员留为 NULL. 我们有一个简单的，no-op release 方法，并且，作为第一个(并且唯一)总线，它的名子时 ldd0. 这个总线设备被注册，使用:

```
ret = device_register(&ldd_bus);
if (ret)
 printk(KERN_NOTICE "Unable to register ldd0\n");
```

一旦调用完成，新总线可在 sysfs 中 /sys/devices 下面见到. 任何加到这个总线的设备接着在 /sys/devices/ldd0 下显示.

14.4.2.2. 设备属性

sysfs 中的设备入口可有属性. 相关的结构是:

```
struct device_attribute {
 struct attribute attr;
 ssize_t (*show)(struct device *dev, char *buf);
 ssize_t (*store)(struct device *dev, const char *buf,
 size_t count);
};
```

这些属性结构可在编译时建立，使用这些宏:

```
DEVICE_ATTR(name, mode, show, store);
```

结果结构通过前缀 dev_attr_ 到给定名子上来命名. 属性文件的实际管理使用通常的函数对来处理:

```
int device_create_file(struct device *device, struct device_attribute *entry);
void device_remove_file(struct device *dev, struct device_attribute *attr);
```

struct bus_type 的 dev_attrs 成员指向一个缺省的属性列表，这些属性给添加到总线的每个设备创建.

14.4.2.3. 设备结构嵌入

设备结构包含设备模型核心需要的来模型化系统的信息. 大部分子系统，但是，跟踪关于它们驻留的设备的额外信息. 结果，对设备很少由空设备结构所代表；相反，这个结构，如同 kobject 结构，常常是嵌入一个更高级的设备表示中. 如果你查看 struct pci_dev

的定义或者 struct usb_device 的定义, 你会发现一个 struct device 埋在其中. 常常地, 低层驱动甚至不知道 struct device, 但是有例外.

lddbus 驱动创建它自己的设备类型(struct ldd_device) 并且期望单独的设备驱动来注册它们的设备使用这个类型. 它是一个简单结构:

```
struct ldd_device {
 char *name;
 struct ldd_driver *driver;
 struct device dev;
};
#define to_ldd_device(dev) container_of(dev, struct ldd_device, dev);
```

这个结构允许驱动提供一个实际的名子给设备(这可以清楚地不同于它的总线 ID, 存储于设备结构) 以及一个这些驱动信息的指针. 给真实设备的结构常常还包含关于供应者信息, 设备型号, 设备配置, 使用的资源, 等等. 可以在 struct pci_dev (<linux/pci.h>) 或者 struct usb_device (<linux/usb.h>) 中找到好的例子. 一个方便的宏 (to_ldd_device) 也为 struct ldd_device 定义, 使得容易转换指向被嵌入的结构的指针为 ldd_device 指针.

lddbus 输出的注册接口看来如此:

```
int register_ldd_device(struct ldd_device *ldddev)
{
 ldddev->dev.bus = &ldd_bus_type;
 ldddev->dev.parent = &ldd_bus;
 ldddev->dev.release = ldd_dev_release;
 strncpy(ldddev->dev.bus_id, ldddev->name, BUS_ID_SIZE);
 return device_register(&ldddev->dev);

}
EXPORT_SYMBOL(register_ldd_device);
```

这里, 我们简单地填充一些嵌入的设备结构成员(单个驱动不应当需要知道这个), 并且注册这个设备到驱动核心. 如果我们想添加总线特定的属性到设备, 我们可在这里做.

为显示这个接口如何使用, 我们介绍另一个例子驱动, 我们称为 sculld. 它是在第 8 章介绍的 scullp 驱动上的另一个变体. 它实现通用的内存区设备, 但是 sculld 也使用 Linux 设备模型, 通过 lddbus 接口.

sculld 驱动添加一个它自己的属性到它的设备入口; 这个属性, 称为 dev, 仅仅包含关联的设备号. 这个属性可被一个模块用来加载脚本或者热插拔子系统, 来自动创建设备节点, 当设备被添加到系统时. 这个属性的设置遵循常用模式:

```
static ssize_t sculld_show_dev(struct device *ddev, char *buf)
{
 struct sculld_dev *dev = ddev->driver_data;
 return print_dev_t(buf, dev->cdev.dev);
}
```

```
static DEVICE_ATTR(dev, S_IRUGO, sculld_show_dev, NULL);
```

接着，在初始化时间，设备被注册，并且 dev 属性被创建通过下面的函数:

```
static void sculld_register_dev(struct sculld_dev *dev, int index)
{
 sprintf(dev->devname, "sculld%d", index);
 dev->ldev.name = dev->devname;
 dev->ldev.driver = &sculld_driver;
 dev->ldev.dev.driver_data = dev;
 register_ldd_device(&dev->ldev);
 device_create_file(&dev->ldev.dev, &dev_attr_dev);
}
```

注意，我们使用 driver_data 成员来存储指向我们自己的内部的设备结构的指针.

14.4.3. 设备驱动

设备模型跟踪所有对系统已知的驱动. 这个跟踪的主要原因是使驱动核心能匹配驱动和新设备. 一旦驱动在系统中是已知的对象，但是，许多其他的事情变得有可能. 设备驱动可输出和任何特定设备无关的信息和配置变量，例如:

驱动由下列结构定义:

```
struct device_driver {
 char *name;
 struct bus_type *bus;
 struct kobject kobj;
 struct list_head devices;
 int (*probe)(struct device *dev);
 int (*remove)(struct device *dev);
 void (*shutdown) (struct device *dev);
};
```

再一次，几个结构成员被忽略(全部内容见 <linux/device.h>). 这里，name 是驱动的名子(它在 sysfs 中出现)，bus 是这个驱动使用的总线类型，kobj 是必然的 kobject，devices 是当前绑定到这个驱动的所有设备的列表，probe 是一个函数被调用来查询一个特定设备的存在(以及这个驱动是否可以使用它)，remove 当设备从系统中去除时被调用，shutdown 在关闭时被调用来关闭设备.

使用 device_driver 结构的函数的形式，现在应当看来是类似的(因此我们快速涵盖它们). 注册函数是:

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

通常的属性结构在:

```
struct driver_attribute {
 struct attribute attr;
```

```
 ssize_t (*show)(struct device_driver *drv, char *buf);
 ssize_t (*store)(struct device_driver *drv, const char *buf,
 size_t count);
};
DRIVER_ATTR(name, mode, show, store);
```

以及属性文件以通常的方法创建:

```
int driver_create_file(struct device_driver *drv, struct driver_attribute *attr);
void driver_remove_file(struct device_driver *drv, struct driver_attribute *attr);
```

bus_type 结构含有一个成员(drv_attrs) 指向一套缺省属性, 对所有关联到这个总线的驱动都创建.

14.4.3.1. 驱动结构嵌入

如同大部分驱动核心结构的情形, device_driver 结构常常被发现嵌到一个更高级的, 总线特定的结构. lddbus 子系统不会和这样的趋势相反, 因此它已定义了它自己的 ldd_driver 结构:

```
struct ldd_driver {
 char *version;
 struct module *module;
 struct device_driver driver;
 struct driver_attribute version_attr;
};
#define to_ldd_driver(drv) container_of(drv, struct ldd_driver, driver);
```

这里, 我们要求每个驱动提供特定当前软件版本, 并且 lddbus 输出这个版本字串为它知道的每个驱动. 总线特定的驱动注册函数是:

```
int register_ldd_driver(struct ldd_driver *driver)
{

 int ret;
 driver->driver.bus = &ldd_bus_type;
 ret = driver_register(&driver->driver);
 if (ret)
 return ret;
 driver->version_attr.attr.name = "version";
 driver->version_attr.attr.owner = driver->module;
 driver->version_attr.attr.mode = S_IRUGO;
 driver->version_attr.show = show_version;
 driver->version_attr.store = NULL;
 return driver_create_file(&driver->driver, &driver->version_attr);
}
```

这个函数的第一部分只注册低级的 device_driver 结构到核心; 剩下的建立版本属性. 因为这个属性在运行时被创建, 我们不能使用 DRIVER_ATTR 宏; 反之,

driver_attribute 结构必须手工填充. 注意我们设定属性的拥有者为驱动模块, 不是 lddbus 模块; 这样做的理由是可以在为这个属性的 show 函数的实现中见到:

```
static ssize_t show_version(struct device_driver *driver, char *buf)
{
 struct ldd_driver *ldriver = to_ldd_driver(driver);
 sprintf(buf, "%s\n", ldriver->version);
 return strlen(buf);
}
```

有人可能认为属性拥有者应当是 lddbus 模块, 因为实现这个属性的函数在那里定义. 这个函数, 但是, 是使用驱动自身所创建的 ldd_driver 结构. 如果那个结构在一个用户空间进程试图读取版本号时要消失, 事情会变得麻烦. 指定驱动模块作为属性的拥有者阻止了模块被卸载, 在用户空间保持属性文件打开时. 因为每个驱动模块创建一个对 lddbus 模块的引用, 我们能确信 lddbus 不会在一个不合适的时间被卸载.

为完整起见, sculld 创建它的 ldd_driver 结构如下:

```
static struct ldd_driver sculld_driver = { .version = "$Revision: 1.1
$", .module = THIS_MODULE, .driver = { .name = "sculld", }, };
```

一个简单的对 register_ldd_driver 的调用添加它到系统中. 一旦完成初始化, 驱动信息可在 sysfs 中见到:

```
$ tree /sys/bus/ldd/drivers
/sys/bus/ldd/drivers
`-- sculld
 |-- sculld0 -> ../../../../devices/ldd0/sculld0
 |-- sculld1 -> ../../../../devices/ldd0/sculld1
 |-- sculld2 -> ../../../../devices/ldd0/sculld2
 |-- sculld3 -> ../../../../devices/ldd0/sculld3
 `-- version
```

---

[46] 这个总线的逻辑名子, 当然, 应当是"sbus", 但是这个名子已经被一个真实的, 物理总线采用.

14.5. 类

我们在本章中要考察最后的设备模型概念是类. 一个类是一个设备的高级视图, 它抽象出低级的实现细节. 驱动可以见到一个 SCSI 磁盘或者一个 ATA 磁盘, 在类的级别, 它们都是磁盘. 类允许用户空间基于它们做什么来使用设备, 而不是它们如何被连接或者它们如何工作.

几乎所有的类都在 sysfs 中在 /sys/class 下出现. 因此, 例如, 所有的网络接口可在 /sys/class/net 下发现, 不管接口类型. 输入设备可在 /sys/class/input 下, 以及串行设备在 /sys/class/tty. 一个例外是块设备, 由于历史的原因在 /sys/block.

类成员关系常常由高级的代码处理，不必要驱动的明确的支持. 当 sbull 驱动(见 16 章) 创建一个虚拟磁盘设备，它自动出现在 /sys/block. snull 网络驱动(见 17 章)没有做任何特殊事情给它的接口在 /sys/class/net 中出现. 将有多次，但是，当驱动结束直接处理类.

在许多情况，类子系统是最好的输出信息到用户空间的方法. 当一个子系统创建一个类，它完全拥有这个类，因此没有必要担心哪个模块拥有那里发现的属性. 它也用极少的时间徘徊于更加面向硬件的 sysfs 部分来了解，它不是一个直接浏览的好地方. 用户会更加高兴地在 /sys/class/some-widget 中发现信息，而不是，/sys/device/pci0000:00/0000:00:10.0/usb2/2-0:1.0.

驱动核心输出 2 个清晰的接口来管理类. class_simple 函数设计来尽可能容易地添加新类到系统. 它们的主要目的，常常，是暴露包含设备号的属性来使能设备节点的自动创建. 常用的类接口更加复杂但是同时提供更多特性. 我们从简单版本开始.

14.5.1. class_simple 接口

class_simple 接口意图是易于使用，以至于没人会抱怨没有暴露至少一个包含设备的被分配的号的属性. 使用这个接口只不过是一对函数调用，没有通常的和 Linux 设备模型关联的样板.

第一步是创建类自身. 使用一个对 class_simple_create 的调用来完成:

```
struct class_simple *class_simple_create(struct module *owner, char *name);
```

这个函数使用给定的名子创建一个类. 这个操作可能失败，当然，因此在继续之前返回值应当一直被检查(使用 IS_ERR，在第 1 章的"指针和错误值"一节中描述过).

一个简单的类可被销毁，使用:

```
void class_simple_destroy(struct class_simple *cs);
```

创建一个简单类的真实目的是添加设备给它; 这个任务使用:

```
struct class_device *class_simple_device_add(struct class_simple *cs, dev_t devnum, struct device *device, const char *fmt, ...);
```

这里，cs 是之前创建的简单类，devnum 是分配的设备号，device 是代表这个设备的 struct device，其他的参数是一个 printk-风格 的格式串和参数来创建设备名子. 这个调用添加一项到类，包含一个属性，dev，含有设备号. 如果设备参数是非 NULL，一个符号连接(称为 device)指向在 /sys/devices 下的设备的入口.

可能添加其他的属性到设备入口. 它只是使用 class_device_create_file，我们在下一节和完整类子系统所剩下的内容讨论.

当设备进出时类产生热插拔事件. 如果你的驱动需要添加变量到环境中给用户空间事件处理者，可以建立一个热插拔回调，使用:

```
int class_simple_set_hotplug(struct class_simple *cs,
 int (*hotplug)(struct class_device *dev,
 char **envp, int num_envp,
 char *buffer, int buffer_size));
```

当你的设备离开时，类入口应当被去除，使用:

```
void class_simple_device_remove(dev_t dev);
```

注意，由 class_simple_device_add 返回的 class_device 结构这里不需要；设备号(它当然应当是唯一的)足够了.

14.5.2. 完整的类接口

class_simple 接口满足许多需要，但是有时需要更多灵活性. 下面的讨论描述如何使用完整的类机制，class_simple 正是基于此. 它是简短的：类函数和结构遵循设备模型其他部分相同的模式，因此这里没有什么真正是新的.

14.5.2.1. 管理类

一个类由一个 struct class 的实例来定义:

```
struct class {
 char *name;
 struct class_attribute *class_attrs;
 struct class_device_attribute *class_dev_attrs;
 int (*hotplug)(struct class_device *dev, char **envp,
 int num_envp, char *buffer, int buffer_size);
 void (*release)(struct class_device *dev);
 void (*class_release)(struct class *class);
 /* Some fields omitted */
};
```

每个类需要一个唯一的名子，它是这个类如何在 /sys/class 中出现. 当这个类被注册，由 class_attrs 所指向的数组中列出的所有属性被创建. 还有一套缺省属性给每个添加到类中的设备；class_dev_attrs 指向它们. 有通常的热插拔函数来添加变量到环境中，当事件产生时. 还有 2 个释放方法：release 在无论何时从类中去除一个设备时被调用，而 class_release 在类自己被释放时调用.

注册函数是:

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

使用属性的接口不应当在这点吓人:

```
struct class_attribute {
 struct attribute attr;
```

```
 ssize_t (*show)(struct class *cls, char *buf);
 ssize_t (*store)(struct class *cls, const char *buf, size_t count);
};
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute *attr);
```

14.5.2.2. 类设备

一个类的真正目的是作为一个是该类成员的设备的容器. 一个成员由 struct class_device 来表示:

```
struct class_device {
struct kobject kobj;
struct class *class;
struct device *dev;
void *class_data;
char class_id[BUS_ID_SIZE];
 };
```

class_id 成员持有设备名子, 如同它在 sysfs 中的一样. class 指针应当指向持有这个设备的类, 并且 dev 应当指向关联的设备结构. 设置 dev 是可选的; 如果它是非 NULL, 它用来创建一个符号连接从类入口到对应的在 /sys/devices 下的入口, 使得易于在用户空间找到设备入口. 类可以使用 class_data 来持有一个私有指针.

通常的注册函数已经被提供:

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
```

类设备接口也允许重命名一个已经注册的入口:

```
int class_device_rename(struct class_device *cd, char *new_name);
```

类设备入口有属性:

```
struct class_device_attribute {
 struct attribute attr;
 ssize_t (*show)(struct class_device *cls, char *buf);
 ssize_t (*store)(struct class_device *cls, const char *buf,
 size_t count);
};

CLASS_DEVICE_ATTR(name, mode, show, store);
int class_device_create_file(struct class_device *cls, const struct class_device_attribute
*attr);
void class_device_remove_file(struct class_device *cls, const struct
class_device_attribute *attr);
```

一个缺省的属性集合，在类的 class_dev_attrs 成员，被创建当类设备被注册时；class_device_create_file 可用来创建额外的属性. 属性还可以被加入到由 class_simple 接口创建的类设备.

14.5.2.3. 类接口

类子系统有一个额外的在 Linux 设备模型其他部分找不到的概念. 这个机制称为一个接口，但是它是，也许，最好作为一种触发机制可用来在设备进入或离开类时得到通知.

一个接口被表示，使用：

```
struct class_interface {
 struct class *class;
 int (*add) (struct class_device *cd);
 void (*remove) (struct class_device *cd);
};
```

接口可被注册或注销，使用：

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```

一个接口的功能是简单明了的. 无论何时一个类设备被加入到在 class_interface 结构中指定的类时，接口的 add 函数被调用. 这个函数可进行任何额外的这个设备需要的设置；这个设置常常采取增加更多属性的形式，但是其他的应用都可能. 当设备被从类中去除，remove 方法被调用来进行任何需要的清理.

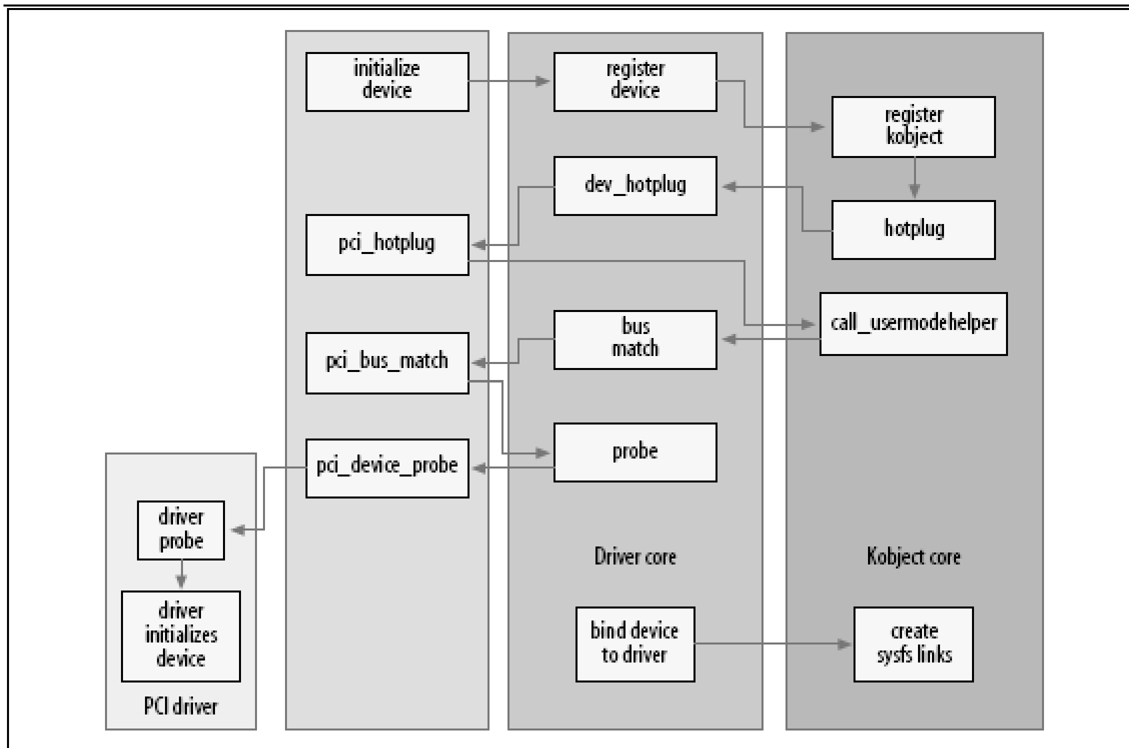
可注册多个接口给一个类.

14.6. 集成起来

为更好理解驱动模型做什么，让我们通览一个设备在内核中的生命周期的阶段. 我们描述 PCI 子系统如何与驱动模型交互，一个驱动如何被加入和去除的基本概念，以及一个设备如何从系统中被加入和去除. 这些细节，即便特别地描述 PCI 内核代码，适用所有其他的使用驱动核心来管理它们的驱动和设备的子系统.

PCI 核心，驱动核心和单独的 PCI 驱动之间的交互是非常复杂，如同图 创建设备过程所示.

图 14.3. 创建设备过程



14.6.1. 添加一个设备

PCI 子系统声明一个单个 struct bus_type 称为 pci_bus_type, 它使用下列值初始化:

```
struct bus_type pci_bus_type = {
 .name  = "pci",
 .match  = pci_bus_match,
 .hotplug  = pci_hotplug,
 .suspend  = pci_device_suspend,
 .resume  = pci_device_resume,
 .dev_attrs = pci_dev_attrs, };
```

这个 pci_bus_type 变量被注册到驱动内核, 当 PCI 子系统通过对 bus_register 的调用被加载入内核时. 当这个发生时, 驱动核心创建一个 sysfs 目录在 /sys/bus/pci 里, 它包含 2 个目录: devices 和 drivers.

所有的 PCI 驱动必须定义一个 struct pci_driver 变量, 它定义了这个 PCI 驱动能够做的不同的功能(更多的关于 PCI 子系统和如何编写一个 PCI 驱动的信息, 见 12 章). 那个结构包含一个 struct device_driver, 它接着被 PCI 核心初始化, 当 PCI 驱动被注册时.

```
/* initialize common driver fields */
drv->driver.name = drv->name;
drv->driver.bus = &pci_bus_type;
drv->driver.probe = pci_device_probe;
drv->driver.remove = pci_device_remove;
drv->driver.kobj.ktype = &pci_driver_kobj_type;
```

这个代码为驱动建立总线来指向 pci_bus_type 以及使 probe 和 remove 函数来指向 PCI 核心内的函数. 驱动的 kobject 的 ktype 被设置为变量 pci_driver_kobj_type, 为使 PCI 驱动的属性文件正常工作. 接着 PCI 核心注册 PCI 驱动到驱动核心:

```
/* register with core */
error = driver_register(&drv->driver);
```

驱动现在准备好被绑定到任何一个它支持的 PCI 设备.

PCI 核心, 在来自特定结构的实际和 PCI 总线交谈的代码的帮助下, 开始探测 PCI 地址空间, 查找所有的 PCI 设备. 当一个 PCI 设备被发现, PCI 核心在内存中创建一个 struct pci_dev 类型的新变量. struct pci_dev 结构的一部分看来如下:

```
struct pci_dev {
 /* ... */
 unsigned int devfn;
 unsigned short vendor;
 unsigned short device;
 unsigned short subsystem_vendor;
 unsigned short subsystem_device;
 unsigned int class;
 /* ... */
 struct pci_driver *driver;
 /* ... */
 struct device dev;
 /* ... */

};
```

这个 PCI 设备的总线特定的成员被 PCI 核心初始化(devfn, vendor, device, 和其他成员), 并且 struct device 变量的 parent 变量被设置为这个 PCI 设备所在的 PCI 总线设备. bus 变量被设置指向 pci_bus_type 结构. 接下来 name 和 bus_id 变量被设置, 根据读自 PCI 设备的 name 和 ID.

在 PCI 设备结构被初始化之后, 设备被注册到驱动核心, 使用:

```
device_register(&dev->dev);
```

在 device_register 函数中, 驱动核心初始化设备的许多成员, 注册设备的 kobject 到 kobject 核心(它导致一个热插拔事件产生, 但是我们在本章后面讨论), 接着添加设备到驱动的 parent 所持有的设备列表中. 完成这个使所有的设备可被以正确的顺序浏览, 一直知道每一个位于设备层次中哪里.

设备接着被添加到所有设备的总线特定的列表中, 在本例中, pci_bus_type 列表. 接着注册到这个总线的所有驱动的列表被检查, 并且总线的匹配功能被调用给每个驱动, 指定这个设备. 对于 pci_bus_type 总线, 匹配函数被 PCI 核心设定为指向 pci_bus_match 函数, 在设备被提交给驱动核心前.

pci_bus_match 函数转换驱动核心传递给它的 struct device 为一个 struct pci_dev. 它还转换 struct device_driver 为一个 struct pci_driver , 并接着查看设备的 PCI 设备特定信息和驱动, 看是否这个驱动声明它能够支持这类设备. 如果匹配不成功, 函数返回 0 给驱动核心, 并且驱动核心移向列表中的下一个驱动.

如果匹配成功, 函数返回 1 给驱动核心. 这使驱动核心设置 struct device 中的驱动指针指向这个驱动, 并且接着调用在 struct device_driver 中特定的 probe 函数.

早些时候, 在 PCI 驱动注册到驱动核心之前, probe 变量被设为指向 pci_device_probe 函数. 这个函数转换(又一次) struct device 为一个 struct pci_dev, 在设备中设置的 struct driver 为一个 struct pci_driver. 它再次验证这个驱动声明它可以支持这个设备(这意味着一个重复的额外检查, 某些未知的原因), 递增设备的引用计数, 并且接着调用 PCI 驱动的 probe 函数, 用一个指向它应当被绑定到的 struct pci_dev 结构的指针.

如果这个 PCI 驱动的 probe 函数认为它不能处理这个设备由于某些原因, 它返回一个负的错误值, 这个值被传递回驱动核心并且使它继续深入设备列表来和这个设备匹配一个. 如果这个 probe 函数能够认领这个设备, 它做所有的需要的初始化来正确处理这个设备, 并且接着它返回 0 给驱动核心. 这使驱动核心来添加设备到当前被这个特定驱动所绑定的所有设备列表, 并且创建一个符号连接到这个它现在控制的设备, 在这个驱动在 sysfs 的目录. 这个符号连接允许用户准确见到哪个设备被绑定到哪个设备. 这可被见到, 如:

```
$ tree /sys/bus/pci
/sys/bus/pci/
|-- devices
|   |-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|   |-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
|   |-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|   |-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|   |-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|   |-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
|   |-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|   |-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|   |-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|   |-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|   |-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|   |-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|   |-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|   |-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|   |-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
|   `-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
`-- drivers
 |-- ALI15x3_IDE
 | `-- 0000:00:0f.0 -> ../../../../devices/pci0000:00/0000:00:0f.0
 |-- ehci_hcd
 | `-- 0000:00:09.2 -> ../../../../devices/pci0000:00/0000:00:09.2
 |-- ohci_hcd
 | |-- 0000:00:02.0 -> ../../../../devices/pci0000:00/0000:00:02.0
 | |-- 0000:00:09.0 -> ../../../../devices/pci0000:00/0000:00:09.0
 | `-- 0000:00:09.1 -> ../../../../devices/pci0000:00/0000:00:09.1
```

```
|-- orinoco_pci
| `-- 0000:00:12.0 -> ../../../../devices/pci0000:00/0000:00:12.0
|-- radeonfb
| `-- 0000:00:14.0 -> ../../../../devices/pci0000:00/0000:00:14.0
|-- serial
`-- trident
`-- 0000:00:04.0 -> ../../../../devices/pci0000:00/0000:00:04.0
```

14.6.2. 去除一个设备

一个 PCI 可用多个不同的方法被从系统中去除. 所有的 card-bus 设备在一个不同的物理因素上是真正的 PCI 设备, 并且内核 PCI 核心不区分它们. 允许在机器运行时加减 PCI 设备的系统正变得更加普遍, 并且 Linux 支持它们. 还有一个伪 PCI 热插拔驱动允许开发者来测试看是否他们的 PCI 驱动正确处理系统运行中的设备去除. 这个模块称为 fakephp 并且使内核认为 PCI 设备已消失, 但是它不允许用户物理上从系统中去除一个 PCI 设备, 这个系统没有合适的硬件来这样做. 见这个驱动的文档来获取更多关于如何使用它测试你的 PCI 驱动的信息.

PCI 核心发挥了不少于它增加设备的努力到去除它. 当一个 PCI 设备要被去除, pci_remove_bus_device 函数被调用. 这个函数做一些 PCI-特定 的清理和日常工作, 并且接着使用一个指向 struct pci_dev 的 struct device 成员的指针调用 device_unregister 函数.

在 device_unregister 函数中, 驱动核心只从绑定到这个设备(如果有)的驱动解除连接 sysfs 文件, 从它的内部设备列表中去除这个设备, 并且使用指向包含在 struct device 结构中的 struct kobject 的指针调用 kobject_del. 这个函数用一个 hotplug 调用到用户空间来声明 kobject 现在被从系统中去除, 并且接着它删除所有的和 kobject 关联的 sysfs 文件以及这个 kobject 起初已创建的 sysfs 目录自身.

kobject_del 函数也去除设备自身的 kobject 引用. 如果那个引用是最后一个(意味着没有用户空间文件为这个 sysfs 的设备入口而打开), 接着是 PCI 设备自身的 release 函数, pci_release_dev, 被调用. 这个函数只释放 struct pci_dev 占用的内存.

此后, 所有的和这个设备关联的 sysfs 入口被去除, 并且和这个设备关联的内存被释放. PCI 设备现在完全从系统中被去除.

14.6.3. 添加一个驱动

一个 PCI 驱动被添加到 PCI 核心, 当它调用 pci_register_driver 函数时. 这个函数只初始化 struct device_driver 结构, 这个结构包含在 struct pci_driver 结构里面, 如同之前在关于添加设备的一节中提过的. 接着 PCI 核心使用指向包含在 struct pci_driver 结构中的 sturct device_driver 结构的指针调用在驱动核心的 driver_register 函数.

driver_register 函数初始化在 struct device_driver 结构中的几个锁, 并且接着调用 bus_add_driver 函数. 这个函数进行下面的步骤:

- 查找驱动要被关联的总线. 如果这个总线被发现, 函数立刻返回.

- 驱动的 sysfs 目录被创建，基于驱动的名子和它被关联的总线.
- 总线的内部锁被获取，接着所有的已经注册到总线的设备被检查，匹配函数为它们被调用，就象当一个新设备被添加时. 如果那个匹配函数成功，接着剩下的绑定过程发生，如同在前面章节描述过的.

14.6.4. 去除一个驱动

去除一个驱动是一个非常容易的动作. 对于一个 PCI 驱动，驱动调用 pci_unregister_driver 函数. 这个函数只调用驱动核心函数 driver_unregister，使用一个指向传递给它的 struct pci_driver 的 struct devie_driver 的指针.

deiver_unregister 函数处理一些基本的日常工作，通过清理某些在 sysfs 树中连接到这个驱动入口的 sysfs 属性. 它接着列举所有的连接到这个驱动的设备并且为它调用 release 函数. 发生这个恰好象前面提过的 release 函数，当一个设备从系统中去除时.

在所有的设备从驱动中被解绑定后，驱动代码完成这个独特的逻辑:

```
down(&drv->unload_sem);
up(&drv->unload_sem);
```

这就在返回函数的调用者之前完成. 这个锁被获取因为代码需要等待所有的对这个驱动的引用计数在它可安全返回前掉到 0. 需要这样是因为 driver_unregister 函数最普遍被作为一个要卸载的模块退出的路径来调用. 模块需要保留在内存只要驱动被设备引用并且等待这个锁被释放，这允许内核知道当可以安全从内存去除驱动时.

14.7. 热插拔

有 2 个不同方法来看热插拔. 内核看待热插拔为硬件，内核和内核驱动之间的交互. 用户看待热插拔是内核和用户空间的通过称为 /sbin/hotplug 的程序的交互. 这个程序被内核调用，当它想通知用户空间某种热插拔事件刚刚在内核中发生.

14.7.1. 动态设备

术语"热插拔"最普遍使用的意义产生于当讨论这样的事实时，几乎所有的计算机系统现在能够处理当系统有电时设备的出现或消失. 这非常不同于只是几年前的计算机系统，那时程序员知道他们只需要在启动时扫描所有的设备，并且他们从不必担心他们的设备消失直到整个机器被关电. 现在，随着 USB 的出现，CardBus，PCMCIA，IEEE1394，和 PCI 热插拔控制器，Linux 内核需要能够可靠地运行不管什么硬件从系统中增加或去除. 这产生了一个额外的负担给设备驱动作者，因为现在他们必须一直处理一个没有任何通知而突然从地下冒出来的设备.

每个不同的总线类型以不同方式处理一个设备的消失. 例如，当一个 PCI ，CardBus，或者 PCMCIA 设备从系统中去除，在驱动通过它的去除函数被通知之前常常是一会儿. 在发生这个前，所有的从 PCI 的读返回所有的位集合. 这意味着驱动需要一直检查它们从 PCI 总线读取的值并且能够正确处理 0xff 值.

这个的一个例子可在 drivers/usb/host/ehci-hcd.c 驱动中见到, 它是一个 PCI 驱动给一个 UBS 2.0(高速)控制卡. 它有下面的代码在它的主握手循环中来探测是否控制块已经从系统中去除.

```
result = readl(ptr);
if (result == ~(u32)0)  /* card removed */
 return -ENODEV;
```

对于 USB 驱动, 当一个 USB 驱动被绑定到的设备被从系统中去除, 任何挂起的已被提交给设备的 urbs 以错误 -ENODEV 失败. 如果发生这个情况, 驱动需要识别这个错误并且正确清理任何挂起的 I/O .

可热插拔的设备不只限于传统的设备, 例如鼠标, 键盘, 和网卡. 有大量的系统现在支持整个 CPU 和内存条的移出. 幸运地, Linux 内核正确处理这些核心"系统"设备的加减, 以至于单个设备驱动不需要注意这些事情.

14.7.2. /sbin/hotplug 工具

如同本章中前面提过的, 无论何时一个设备从系统中增删, 都产生一个"热插拔事件". 这意味着内核调用用户空间程序 /sbin/hotplug. 这个程序典型地是一个非常小的 bash 脚本, 只传递执行给一系列其他的位于 /etc/hot-plug.d/ 目录树的程序. 对于大部分的 Linux 发布, 这个脚本看来如下:

```
DIR="/etc/hotplug.d"
for I in "${DIR}/$1/"*.hotplug "${DIR}/"default/*.hotplug ; do
 if [ -f $I ]; then
 test -x $I && $I $1 ;
 fi
done
exit 1
```

换句话说, 这个脚本搜索所有的有 .hotplug 后缀的可能对这个事件感兴趣的程序并调用它们, 传递给它们许多不同的环境变量, 这些环境变量已经被内核设置. 更多关于 /sbin/hotplug 脚本如何工作的细节可在程序的注释中找到, 以及在 hotplug(8)手册页中.

如同前面提到的, /sbin/hotplug 被调用无论何时一个 kobject 被创建或销毁. 热插拔程序被用一个提供事件名子的单个命令行参数调用. 核心内核和涉及到的特定子系统也设定一系列带有关于发生了什么的信息的环境变量(下面描述). 这些变量被热插拔程序使用来判定刚刚在内核发生了什么, 以及是否有任何特定的动作应当采取.

传递给 /sbin/hotplug 的命令行参数是关联这个热插拔事件的名子, 如同分配给 kobject 的 kset 所决定的. 这个名子可通过一个对属于本章前面描述过的 kset 的 hotplug_ops 结构的 name 函数的调用来设定; 如果那个函数不存在或者从未被调用, 名子是 kset 自身的名子.

一直为 /sbin/hotplug 设定的缺省的环境变量是:

ACTION

这个字符串 add 或 remove, 只根据是否这个对象是被创建或者销毁.

DEVPATH

一个目录路径, 在 sysfs 文件系统中, 它指向在被创建或销毁的 kobject. 注意 sysfs 文件系统的加载点不是添加到这路径, 因此是由用户空间程序来决定这个.

SEQNUM

这个热插拔事件的顺序号. 顺序号是一个 64-位 数, 它每次产生热插拔事件都递增. 这允许用户空间以内核产生它们的顺序来排序热插拔事件, 因为对一个用户空间程序可能乱序运行.

SUBSYSTEM

同样的字符串作为前面描述的命令行参数传递.

许多不同的总线子系统都添加它们自己的环境变量到 /sbin/hotplug 调用中, 当关联到总线的设备被添加或从系统中去除. 它们在它们的热插拔回调中做这个, 这个回调在分配给它们的总线(如同在"热插拔操作"一节中描述的)的 struct kset_hotplug_ops 中指定. 这允许用户空间能够自动加载必要的可能需要来控制这个被总线发现的设备的模块. 这里是一个不同总线类型的列表以及它们添加到 /sbin/hotplug 调用中的环境变量.

14.7.2.1. IEEE1394(火线)

任何在 IEEE1394 总线, 也是火线, 上的设备, 由 /sbin/hotplug 参数名和 SUBSYSTEM 环境变量设置为值 ieee1394. ieee1394 子系统也总是添加下列 4 个环境变量:

VENDOR_ID

IEEE1394 的 24-位 供应者 ID.

MODEL_ID

IEEE1394 的 24-位型号 ID.

GUID

设备的 64-位 GUID.

SPECIFIER_ID

24-位值, 指定设备的协议规格的拥有者.

VERSION

指定设备协议规格的版本的值

14.7.2.2. 网络

所有的网络设备都创建一个热插拔事件，当设备注册或者注销在内核. /sbin/hotplug 调用有参数 name 和 SUBSYSTEM 环境变量设置为 net，并且只添加下列环境变量:

INTERFACE

已经从内核注册或注销的接口的名子. 这个的例子是 lo 和 eth0.

14.7.2.3. PCI 总线

任何在 PCI 总线上的设备有参数 name 和 SUBSYSTEM 环境变量设置为值 pci. PCI 子系统也一直添加下面 4 个环境变量:

PCI_CLASS

设备的 PCI 类号，16 进制.

PCI_ID

设备的 PCI 供应商和设备 ID，16 进制，结合成这样的格式 供应者:设备.

PCI_SUBSYS_ID

PCI 子系统供应商和子系统设备 ID，以 子系统供应者:子系统设备 的格式结合.

PCI_SLOT_NAME

PCI 插口"名"，内核给予这个设备的. 它以这样的格式 域:总线:插口:功能. 一个例子可能是: 0000:00:0d.0.

14.7.2.4. 输入

对所有的输入设备(鼠标，键盘，游戏杆，等等)，一个热插拔事件当设备从内核增减时产生. /sbin/hotplug 参数和 SUBSYSTEM 环境变量被设置为值 input. 输入子系统也总是添加下面的环境变量:

PRODUCT

一个多值字符串，用 16 进制列出值没有前导 0. 它的格式是 bustype:vender:product:version.

下列环境变量可能出现，如果设备支持它:

NAME

输入设备的名子, 如同设备给定的.

PHYS

输入子系统给这个设备的设备的物理地址. 它假定是稳定的, 依赖设备所插入的总线的位置.

EV
KEY
REL
ABS
MSC
LED
SND
FF

这些都来自输入设备描述符并且被设置为合适的值如果特定的输入设备支持它.

14.7.2.5. USB 总线

任何在 USB 总线上的设备有参数 name 和 SUBSYSTEM 环境变量设置为 usb. USB 子系统也总是一直添加下列的环境变量:

PRODUCT

一个字符串, idVendor/idProduct/bcdDevice 的格式, 来指定这些 USB 设备特定的成员.

TYPE

一个 bDeviceClass/bDeviceSubClass/bDeviceProtocol 格式的字符串, 指定这些 USB 设备特定的成员.

如果 bDeviceClass 成员设置为 0, 下列的环境变量也被设置:

INTERFACE

一个 bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol 格式的字符串, 指定这些 USB 设备特定成员.

如果这个内核建立选项, CONFIG_USB_DEVICEFS, 它选择 usbfs 文件系统来在内核中建立, 被选中, 下列环境变量也被设置:

DEVICE

一个字符串，在设备所在的 usbfs 文件系统中出现．这个字串以 /proc/bus/usb/USB_BUS_NUMBER/USB_DEVICE_NUMBER 的格式，其中 USB_BUS_NUMBER 是这个设备所在的 USB 总线的 3 个数，USB_DEVICE_NUMBER 是已由内核分配给 USB 设备的 3 位数．

14.7.2.6. SCSI 总线

所有的 SCSI 设备创建一个热插拔事件当 SCSI 设备从内核中创建或去除． /sbin/hotplug 调用有参数 name 和 SUBSYSTEM 环境变量设置为 scsi 给每个添加或去除自系统的 SCSI 设备．没有额外的环境变量由 SCSI 系统添加，但是它被在此提及因为有一个 SCSI 特定的用户空间脚本来决定什么 SCSI 驱动(磁盘，磁带，通用，等等)应当给这个特定 SCSI 设备加载．

14.7.2.7. 膝上电脑坞站

如果一个支持即插即用的膝上电脑坞站被从运行中的 Linux 系统中添加或去除(通过插入膝上电脑到坞站中，或者去除它)，一个热插拔事件被产生．/sbin/hotplug 调用有参数 name 和 SUBSYSTEM 环境变量设为 dock．没有其他的环境变量被设置．

14.7.2.8. S/390 和 zSeries

在 S/390 体系中，通道总线结构支持很广范围的硬件，所有产生 /sbin/hotplug 事件当它们从 Linux 虚拟系统被添加或去除时的硬件．这些设备都有 /sbin/hotplug 参数 name 和 SUBSYSTEM 环境变量设置为 dasd．没有其他环境变量被设置．

14.7.3. 使用 /sbin/hotplug

现在 Linux 内核在调用 /sbin/hotplug 为每个设备，添加和删除自内核，许多非常有用的工具在用户空间已被创建来利用这一点．2 个最常用的工具是 Linux 热插拔脚本和 udev．

14.7.3.1. Linux 热插拔脚本

Linux 热插拔脚本作为 /sbin/hotplug 调用的第一个用户而启动．这些脚本查看内核设置的来描述刚刚发现的设备的不同的环境变量，并接着试图发现一个匹配这个设备的内核模块．

如同前面描述的，当一个驱动使用 MODULE_DEVICE_TABLE 宏，程序 depmod 采用这个信息并创建位于 /lib/module/KERNEL_VERSION/modules.*map 的文件．这个 * 是不同的，根据驱动支持的总线类型．当前，模块 map 文件为使用设备的驱动而产生，这些设备支持 PCI，USB，IEEE1394，INPUT，ISAPNP，和 CCW 子系统．

热插拔脚本使用这些模块映射文本文件，来决定试图加载什么模块来支持内核刚刚发现的设备．它们加载所有的模块，在第一次匹配时不停止，为了使内核发现那个模块工作得最

好. 这些脚本不加载任何模块当驱动被去除时. 如果它们要试图做这个, 它们可能偶然地关闭被同一个要被去除的驱动控制的设备.

注意, 现在 modprobe 程序能直接从模块中读 MODULE_DEVICE_TABLE 信息而不需要模块 map 文件, 热插拔脚本可能被删减为一个小的在 modprobe 程序周围的包装.

14.7.3.2. udev 啥?

在内核中创建统一的驱动模型的一个主要原因是允许用户空间动态管理 /dev 树. 这之前已使用 devfs 的实现在用户空间实现, 但是那个代码底线已慢慢消失, 由于缺少一个活跃的维护者以及一些无法修正的核心 bug. 许多内核开发者认识到如果所有的设备信息被输出给用户空间, 它可能进行所有的必要的 /dev 树的管理.

devfs 在它的设计中有一些非常基础的缺陷. 它需要每个设备驱动被修改来支持它, 并且它要求设备驱动来指定名子和在它所在的 /dev 树中的位置. 它也没有正确处理动态主次编号, 并且它不允许用户空间以简单方式覆盖设备的命名, 这样来强制设备命名策略于内核中而不是在用户空间. Linux 内核开发中非常厌恶使策略在内核中, 并且因为 devfs 命名策略不遵循 Linux 标准基础规格, 它确实困扰他们.

随着 Linux 内核开始安装到大型服务器, 许多用户遇到如何管理大量设备的问题. 超过 10,000 个单一设备的磁盘驱动阵列提出了非常困难的任务, 保证一个特定磁盘一直使用相同的名子命名, 不管它在磁盘阵列的哪里或者它什么时候被内核发现. 同样的问题也折磨着桌面用户, 想插入 2 个 USB 打印机到他们的系统, 并且接着发现它们没有办法保证已知为 /dev/lpt0 的打印机不会改变并分配给其他的打印机如果系统重启.

因此, udev 被创建. 它依靠所有通过 sysfs 输出给用户空间的设备信息, 并且依靠被 /sbin/hotplug 通知有设备添加或去除. 策略决策, 例如给一个设备什么名子, 可在用户空间指定, 内核之外. 这保证了命名策略被从内核中去除并且允许大量每个设备名子的灵活性.

对更多的关于如何使用 udev 和如何配置它的信息, 请看在你的发布中和 udev 软件包一起的文档.

所有的一个设备驱动需要做的, 为 udev 正确使用它, 是确保任何分配给一个驱动控制的设备的主次编号通过 sysfs 输出到用户空间. 对任何使用一个子系统来安排它一个主次编号的驱动, 这已经由子系统完成, 并且驱动不必做任何工作. 做这个的子系统的例子是 tty, misc, usb, input, scsi, block, i2c, network, 和 frame buffer 子系统. 如果你的驱动自己获得一个主次编号, 通过对 cdev_init 函数的调用或者更老的 register_chrdev 函数, 驱动需要被修改以便 udev 能够正确使用它.

udev 查找一个称为 dev 的文件在 sysfs 的 /class/ 树中, 为了决定分配什么主次编号给一个特定设备当它被内核通过 /sbin/hotplug 接口调用时. 一个设备驱动只要为每个它控制的设备创建这个文件. class_simple 接口常常是最易的做这个的方法.

如同" class_simple 接口"一节中提过的, 使用 class_simple 接口的第一步是调用 class_simple_create 函数来创建一个 struct class_simple.

```
static struct class_simple *foo_class;
...
foo_class = class_simple_create(THIS_MODULE, "foo");
if (IS_ERR(foo_class)) {
 printk(KERN_ERR "Error creating foo class.\n");
 goto error;
}
```

这个代码创建一个目录在 sysfs 中 /sys/class/foo.

无论何时你的驱动发现一个新设备, 并且你如第 3 章描述的分配它一个次编号, 驱动应当调用 class_simple_device_add 函数:

```
class_simple_device_add(foo_class, MKDEV(FOO_MAJOR, minor), NULL, "foo%d", minor);
```

这个代码导致在 /sys/class/foo 创建一个子目录称为 fooN, 这里 N 是这个设备的次编号. 在这个目录里创建有一个文件, dev, 它恰好是 udev 为你的设备创建一个设备节点需要的.

当你的驱动从一个设备解除, 并且你放弃它所依附的次编号, 需要调用 class_simple_device_remove 来去除这个设备的 sysfs 入口.

```
class_simple_device_remove(MKDEV(FOO_MAJOR, minor));
```

之后, 当你的整个驱动被关闭, 需要调用 class_simple_destroy 来去除你起初调用 class_simple_create 创建的 class.

```
class_simple_destroy(foo_class);
```

同样 class_simple_device_add 创建的 dev 文件包括主次编号, 由一个 : 隔开. 如果你的驱动不想使用 class_simple 接口因为你想提供其他在子系统的类目录中的文件, 使用 print_dev_t 函数来正确格式化特定设备的主次编号.

14.8. 处理固件

作为一个驱动作者, 你可能发现你面对一个设备必须在它能支持工作前下载固件到它里面. 硬件市场的许多地方的竞争是如此得强烈, 以至于甚至一点用作设备控制固件的 EEPROM 的成本制造商都不愿意花费. 因此固件发布在随硬件一起的一张 CD 上, 并且操作系统负责传送固件到设备自身.

你可能想解决固件问题使用这样的一个声明:

```
static char my_firmware[] = { 0x34, 0x78, 0xa4, ... };
```

但是, 这个方法几乎肯定是一个错误. 将固件编码到一个驱动扩大了驱动的代码, 使固件升级困难, 并且非常可能产生许可问题. 供应商不可能已经发布固件映象在 GPL 之下,

因此和 GPL-许可的代码混合常常是一个错误. 为此, 包含内嵌固件的驱动不可能被接受到主流内核或者被 Linux 发布者包含.

14.8.1. 内核固件接口

正确的方法是当你需要它时从用户空间获取它. 但是, 请抵制试图从内核空间直接打开包含固件的文件的诱惑; 那是一个易出错的操作, 并且它安放了策略(以一个文件名的形式)到内核. 相反, 正确的方法时使用固件接口, 它就是为此而创建的:

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name,
 struct device *device);
```

调用 request_firmware 要求用户空间定位并提供一个固件映象给内核; 我们一会儿看它如何工作的细节. name 应当标识需要的固件; 正常的用法是供应者提供的固件文件名. 某些象 my_firmware.bin 的名子是典型的. 如果固件被成功加载, 返回值是 0(负责常用的错误码被返回), 并且 fw 参数指向一个这些结构:

```
struct firmware {
 size_t size;
 u8 *data;
};
```

那个结构包含实际的固件, 它现在可被下载到设备中. 小心这个固件是来自用户空间的未被检查的数据; 你应当在发送它到硬件之前运用任何并且所有的你能够想到的检查来说服你自己它是正确的固件映象. 设备固件常常包含标识串, 校验和, 等等; 在信任数据前全部检查它们.

在你已经发送固件到设备前, 你应当释放 in-kernel 结构, 使用:

```
void release_firmware(struct firmware *fw);
```

因为 request_firmware 请求用户空间来帮忙, 它保证在返回前睡眠. 如果你的驱动当它必须请求固件时不在睡眠的位置, 异步的替代方法可能要使用:

```
int request_firmware_nowait(struct module *module,
 char *name, struct device *device, void *context,
 void (*cont)(const struct firmware *fw, void *context));
```

这里额外的参数是 moudle(它将一直是 THIS_MODULE), context (一个固件子系统不使用的私有数据指针), 和 cont. 如果都进行顺利, request_firmware_nowait 开始固件加载过程并且返回 0. 在将来某个时间, cont 将用加载的结果被调用. 如果由于某些原因固件加载失败, fw 是 NULL.

14.8.2. 它如何工作

固件子系统使用 sysfs 和热插拔机制. 当调用 request_firmware, 一个新目录在 /sys/class/firmware 下使用你的驱动的名子被创建. 那个目录包含 3 个属性:

loading

这个属性应当被加载固件的用户空间进程设置为 1. 当加载进程完成, 它应当设为 0. 写一个值 -1 到 loading 会中止固件加载进程.

data

data 是一个二进制的接收固件数据自身的属性. 在设置 loading 后, 用户空间进程应当写固件到这个属性.

device

这个属性是一个符号连接到 /sys/devices 下面的被关联入口项.

一旦创建了 sysfs 入口项, 内核为你的设备产生一个热插拔事件. 传递给热插拔处理者的环境包括一个变量 FIRMWARE, 它被设置为提供给 request_firmware 的名子. 这个处理者应当定位固件文件, 并且拷贝它到内核使用提供的属性. 如果这个文件无法找到, 处理者应当设置 loading 属性为 -1.

如果一个固件请求在 10 秒内没有被服务, 内核就放弃并返回一个失败状态给驱动. 超时周期可通过 sysfs 属性 /sys/class/firmware/timeout 属性改变.

使用 request_firmware 接口允许你随你的驱动发布设备固件. 当正确地集成到热插拔机制, 固件加载子系统允许设备简化工作"在盒子之外" 显然这是处理问题的最好方法.

但是, 请允许我们提出多一条警告: 设备固件没有制造商的许可不应当发布. 许多制造商会同意在合理的条款下许可它们的固件, 如果客气地请求; 一些其他的可能不何在. 无论如何, 在没有许可时拷贝和发布它们的固件是对版权法的破坏并且招致麻烦.

14.9. 快速参考

许多函数在本章中已经被介绍过; 这是它们全部的一个快速总结.

14.9.1. Kobjects 结构

```
#include <linux/kobject.h>
```

包含文件, 包含 kobject 的定义, 相关结构, 和函数.

```
void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

用作 kobject 初始化的函数

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

为 kobjects 管理引用计数的函数.

```
struct kobj_type;
struct kobj_type *get_ktype(struct kobject *kobj);
```

表示一个 kobjct 被嵌入的结构类型. 使用 get_ktype 来获得关联到一个给定 kobject 的 kobj_type.

```
int kobject_add(struct kobject *kobj);
extern int kobject_register(struct kobject *kobj);
void kobject_del(struct kobject *kobj);
void kobject_unregister(struct kobject *kobj);
```

kobject_add 添加一个 kobject 到系统, 处理 kset 成员关系, sysfs 表示, 以及热插拔事件产生. kobject_register 是一个方便函数, 它结合 kobject_init 和 kobject_add. 使用 kobject_del 来去除一个 kobject 或者 kobject_unregister, 它结合了 kobject_del 和 kobject_put.

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

为 ksets 初始化和注册的函数.

```
decl_subsys(name, type, hotplug_ops);
```

易于声明子系统的一个宏.

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys);
void subsys_put(struct subsystem *subsys);
```

对子系统的操作.

14.9.2. sysfs 操作

```
#include <linux/sysfs.h>
```

包含 sysfs 声明的包含文件.

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
void sysfs_remove_link(struct kobject *kobj, char *name);
```

创建和去除和一个 kobject 关联的属性文件的函数.

14.9.3. 总线, 设备, 和驱动

```
int bus_register(struct bus_type *bus);
void bus_unregister(struct bus_type *bus);
```

在设备模型中进行注册和注销总线的函数.

```
int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data, int
(*fn)(struct device *, void *));
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void *data, int
(*fn)(struct device_driver *, void *));
```

列举每个设备和驱动的函数, 特别地, 绑定到给定总线的设备.

```
BUS_ATTR(name, mode, show, store);
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

BUS_ATTR 宏可能用来声明一个 bus_attribute 结构, 它可能接着被添加和去除, 使用上面 2 个函数.

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

处理设备注册的函数.

```
DEVICE_ATTR(name, mode, show, store);
int device_create_file(struct device *device, struct device_attribute *entry);
void device_remove_file(struct device *dev, struct device_attribute *attr);
```

处理设备属性的宏和函数.

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

注册和注销一个设备驱动的函数.

```
DRIVER_ATTR(name, mode, show, store);
int driver_create_file(struct device_driver *drv, struct driver_attribute *attr);
void driver_remove_file(struct device_driver *drv, struct driver_attribute *attr);
```

关联驱动属性的宏和函数.

14.9.4. 类

```
struct class_simple *class_simple_create(struct module *owner, char *name);
void class_simple_destroy(struct class_simple *cs);
struct class_device *class_simple_device_add(struct class_simple *cs, dev_t devnum, struct
device *device, const char *fmt, ...);
void class_simple_device_remove(dev_t dev);
int class_simple_set_hotplug(struct class_simple *cs, int (*hotplug)(struct class_device
*dev, char **envp, int num_envp, char *buffer, int buffer_size));
```

实现 class_simple 接口的函数; 它们管理包含一个 dev 属性和很少其他属性的简单的类入口

```
int class_register(struct class *cls);
```

```
void class_unregister(struct class *cls);
```
注册和注销类.

```
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute *attr);
```
处理类属性的常用宏和函数.

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
int class_device_rename(struct class_device *cd, char *new_name);
CLASS_DEVICE_ATTR(name, mode, show, store);
int class_device_create_file(struct class_device *cls, const struct class_device_attribute *attr);
```
属性类设备接口的函数和宏.

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```
添加一个接口到一个类(或去除它)的函数.

14.9.5. 固件

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name, struct device *device);
int request_firmware_nowait(struct module *module, char *name, struct device *device, void *context, void (*cont)(const struct firmware *fw, void *context));
void release_firmware(struct firmware *fw);
```
属性内核固件加载接口的函数.

第 15 章 内存映射和 DMA

本章研究 Linux 内存管理的部分，重点在对于设备驱动作者有用的技术. 许多类型的驱动编程需要一些对于虚拟内存子系统如何工作的理解；我们在本章涉及到的材料来自手头，而不是象我们曾进入更加复杂和性能关键的子系统一样. 虚拟内存子系统也是 Linux 内核核心的非常有趣的部分，并且因而，值得一见.

本章的材料分为 3 个部分:

- 第一部分涉及 mmap 系统调用的实现，它允许设备内存直接映射到一个用户进程地址空间. 不是所有的设备需要 mmap 支持，但是，对一些，映射设备内存可产生可观的性能提高.
- 我们接着看从其他的方向跨过边界，用对直接存取用户空间的讨论. 相对少驱动需要这个能力；在大部分情况下，内核做这种映射而驱动甚至不知道它. 但是了解如何映射用户空间内存到内核(使用 get_user_pages)会有用.
- 最后一节涵盖直接内存存取(DMA) I/O 操作，它提供给外设对系统内存的直接存取.

当然，所有这些技术需要一个对 Linux 内存管理如何工作的理解，因此我们从对这个子系统的总览开始.

15.1. Linux 中的内存管理

不是描述操作系统的内存管理理论，本节试图指出 Linux 实现的主要特点. 尽管你不必是一位 Linux 虚拟内存专家来实现 mmap，一个对事情如何工作的基本了解是有用的. 下面是一个相当长的对内核使用来管理内存的数据结构的描述. 一旦必要的背景已被覆盖，我们就进入使用这个结构.

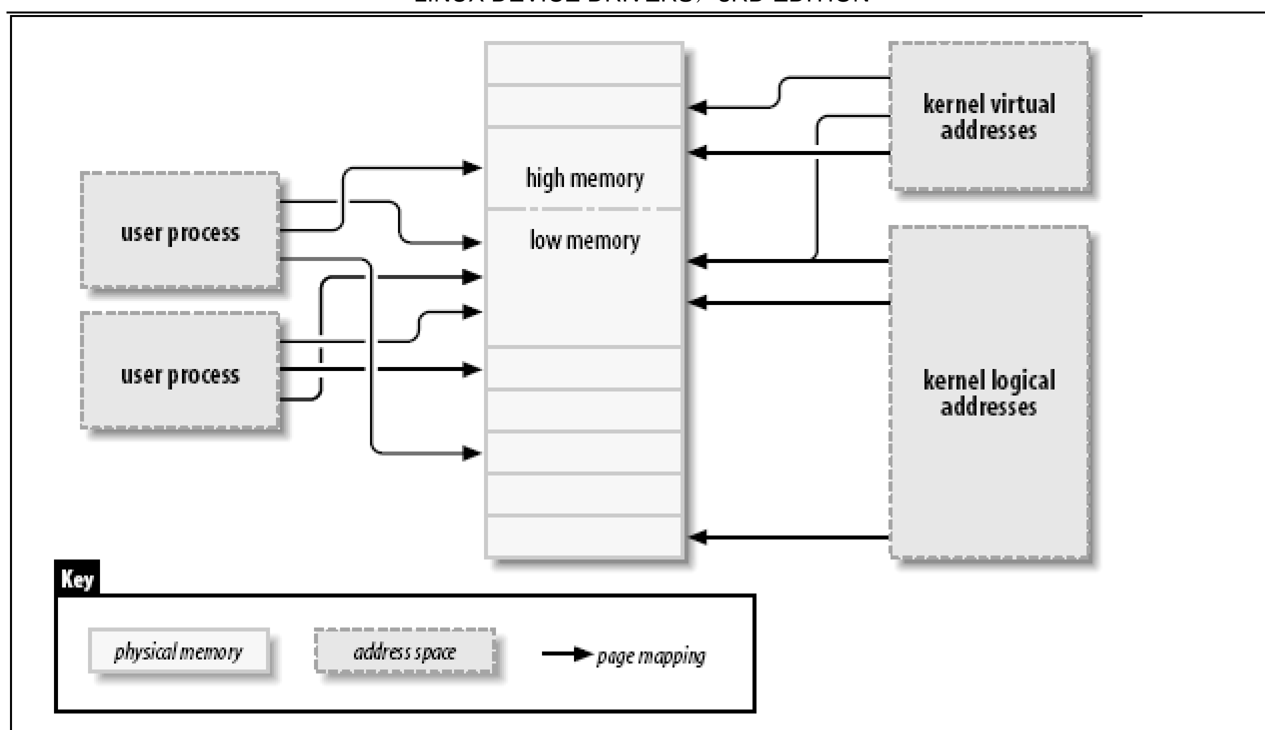
15.1.1. 地址类型

Linux 是，当然，一个虚拟内存系统，意味着用户程序见到的地址不直接对应于硬件使用的物理地址. 虚拟内存引入了一个间接层，它允许了许多好事情. 有了虚拟内存，系统重运行的程序可以分配远多于物理上可用的内存；确实，即便一个单个进程可拥有一个虚拟地址空间大于系统的物理内存. 虚拟内存也允许程序对进程的地址空间运用多种技巧，包括映射成员的内存到设备内存.

至此，我们已经讨论了虚拟和物理地址，但是许多细节被掩盖过去了. Linux 系统处理几种类型的地址，每个有它自己的含义. 不幸的是，内核代码不是一直非常清楚确切地在每个情况下在使用什么类型地地址，因此程序员必须小心.

下面是一个 Linux 中使用的地址类型列表. 图 Linux 中使用的地址类型显示了这个地址类型如何关联到物理内存.

图 15.1. Linux 中使用的地址类型



User virtual addresses

这是被用户程序见到的常规地址. 用户地址在长度上是 32 位或者 64 位, 依赖底层的硬件结构, 并且每个进程有它自己的虚拟地址空间.

Physical addresses

在处理器和系统内存之间使用的地址. 物理地址是 32- 或者 64-位的量; 甚至 32-位系统在某些情况下可使用更大的物理地址.

Bus addresses

在外设和内存之间使用的地址. 经常, 它们和被处理器使用的物理地址相同, 但是这不是必要的情况. 一些体系可提供一个 I/O 内存管理单元(IOMMU), 它在总线和主内存之间重映射地址. 一个 IOMMU 可用多种方法使事情简单(例如, 使散布在内存中的缓冲对设备看来是连续的, 例如), 但是当设定 DMA 操作时对 IOMMU 编程是一个必须做的额外的步骤. 总线地址是高度特性依赖的, 当然.

Kernel logical addresses

这些组成了正常的内核地址空间. 这些地址映射了部分(也许全部)主存并且常常被当作它们是物理内存来对待. 在大部分的体系上, 逻辑地址和它们的相关物理地址只差一个常量偏移. 逻辑地址使用硬件的本地指针大小并且, 因此, 可能不能在重装备的 32-位系统上寻址所有的物理内存. 逻辑地址常常存储于 unsigned long 或者 void * 类型的变量中. 从 kmalloc 返回的内存有内核逻辑地址.

Kernel virtual addresses

内核虚拟地址类似于逻辑地址, 它们都是从内核空间地址到物理地址的映射. 内核虚拟地址不必有逻辑地址空间具备的线性的, 一对一到物理地址的映射, 但是. 所

有的逻辑地址是内核虚拟地址，但是许多内核虚拟地址不是逻辑地址. 例如，vmalloc 分配的内存有虚拟地址(但没有直接物理映射). kmap 函数(本章稍后描述)也返回虚拟地址. 虚拟地址常常存储于指针变量.

如果你有逻辑地址，宏 __pa() (在 <asm/page.h> 中定义)返回它的关联的物理地址. 物理地址可被映射回逻辑地址使用 __va()，但是只给低内存页.

不同的内核函数需要不同类型地址. 如果有不同的 C 类型被定义可能不错，这样请求的地址类型是明确的，但是我们没有这样的好运. 在本章，我们尽力对在哪里使用哪种类型地址保持清晰.

15.1.2. 物理地址和页

物理内存被划分为离散的单元称为页. 系统的许多内部内存处理在按页的基础上完成. 页大小一个体系不同于另一个，尽管大部分系统当前使用 4096-字节的页. 常量 PAGE_SIZE (定义在 <asm/page.h>) 给出了页大小在任何给定的体系上.

如果你查看一个内存地址 - 虚拟或物理 - 它可分为一个页号和一个页内的偏移. 如果使用 4096-字节页，例如，12 位低有效位是偏移，并且剩下的，高位指示页号. 如果你丢弃偏移并且向右移动剩下的部分 offset 位，结果被称为一个页帧号 (PFN). 移位来在页帧号和地址之间转换是一个相当普通的操作. 宏 PAGE_SHIFT 告诉必须移动多少位来进行这个转换.

15.1.3. 高和低内存

逻辑和内核虚拟地址之间的不同在配备大量内存的 32-位系统中被突出. 用 32 位，可能寻址 4 G 内存. 但是，直到最近，在 32-位 系统的 Linux 被限制比那个少很多的内存，因为它建立虚拟地址的方式.

内核(在 x86 体系上，在缺省配置里) 在用户空间和内核之间划分 4-G 虚拟地址; 在 2 个上下文中使用同一套映射. 一个典型的划分分出 3 GB 给用户空间，和 1 GB 给内核空间. [47]内核的代码和数据结构必须要适合这个空间，但是内核地址空间最大的消费者是物理内存的虚拟映射. 内核不能直接操作没有映射到内核的地址空间. 内核，换句话说，需要它自己的虚拟地址给任何它必须直接接触的内存. 因此，多年来，能够被内核处理的的最大量的物理内存是能够映射到虚拟地址的内核部分的数量，减去内核代码自身需要的空间. 结果，基于 x86 的 Linux 系统可以工作在最多稍小于 1 GB 物理内存.

为应对更多内存的商业压力而不破坏 32-位 应用和系统的兼容性，处理器制造商已经增加了"地址扩展"特性到他们的产品中. 结果，在许多情况下，即便 32-位 处理器也能够寻址多于 4GB 物理内存. 但是，多少内存可被直接用逻辑地址映射的限制还存在. 这样内存的最低部分(上到 1 或 2 GB，根据硬件和内核配置)有逻辑地址; 剩下的(高内存)没有. 在存取一个特定高地址页之前，内核必须建立一个明确的虚拟映射来使这个也在内核地址空间可用. 因此，许多内核数据结构必须放在低内存; 高内存用作被保留为用户进程页.

术语"高内存"对有些人可能是疑惑的，特别因为它在 PC 世界里有其他的含义. 因此，为清晰起见，我们将定义这些术语:

Low memory

逻辑地址在内核空间中存在的内存. 在大部分每个系统你可能会遇到, 所有的内存都是低内存.

High memory

逻辑地址不存在的内存, 因为它在为内核虚拟地址设置的地址范围之外.

在 i386 系统上, 低和高内存之间的分界常常设置在刚刚在 1 GB 之下, 尽管那个边界在内核配置时可被改变. 这个边界和在原始 PC 中有的老的 640 KB 限制没有任何关联, 并且它的位置不是硬件规定的. 相反, 它是, 内核自身设置的一个限制当它在内核和用户空间之间划分 32-位地址空间时.

我们将指出使用高内存的限制, 随着我们在本章遇到它们时.

15.1.4. 内存映射和 struct page

历史上, 内核已使用逻辑地址来引用物理内存页. 高内存支持的增加, 但是, 已暴露这个方法的一个明显的问题 -- 逻辑地址对高内存不可用. 因此, 处理内存的内核函数更多在使用指向 struct page 的指针来代替(在 <linux/mm.h> 中定义). 这个数据结构只是用来跟踪内核需要知道的关于物理内存的所有事情.

2.6 内核(带一个增加的补丁)可以支持一个 "4G/4G" 模式在 x86 硬件上, 它以微弱的性能代价换来更大的内核和用户虚拟地址空间.

系统中每一个物理页有一个 struct page. 这个结构的一些成员包括下列:

atomic_t count;

这个页的引用数. 当这个 count 掉到 0, 这页被返回给空闲列表.

void *virtual;

这页的内核虚拟地址, 如果它被映射; 否则, NULL. 低内存页一直被映射; 高内存页常常不是. 这个成员不是在所有体系上出现; 它通常只在页的内核虚拟地址无法轻易计算时被编译. 如果你想查看这个成员, 正确的方法是使用 page_address 宏, 下面描述.

unsigned long flags;

一套描述页状态的一套位标志. 这些包括 PG_locked, 它指示该页在内存中已被加锁, 以及 PG_reserved, 它防止内存管理系统使用该页.

有很多的信息在 struct page 中, 但是它是内存管理的更深的黑魔法的一部分并且和驱动编写者无关.

内核维护一个或多个 struct page 项的数组来跟踪系统中所有物理内存. 在某些系统, 有一个单个数组称为 mem_map. 但是, 在某些系统, 情况更加复杂. 非一致内存存取(NUMA)系统和那些有很大不连续的物理内存的可能有多于一个内存映射数组, 因此打算是可移植的代码在任何可能时候应当避免直接对数组存取. 幸运的是, 只是使用 struct page 指针常常是非常容易, 而不用担心它们来自哪里.

有些函数和宏被定义来在 struct page 指针和虚拟地址之间转换:

struct page *virt_to_page(void *kaddr);

这个宏, 定义在 <asm/page.h>, 采用一个内核逻辑地址并返回它的被关联的 struct page 指针. 因为它需要一个逻辑地址, 它不使用来自 vmalloc 的内存或者高内存.

struct page *pfn_to_page(int pfn);

为给定的页帧号返回 struct page 指针. 如果需要, 它在传递给 pfn_to_page 之前使用 pfn_valid 来检查一个页帧号的有效性.

void *page_address(struct page *page);

返回这个页的内核虚拟地址, 如果这样一个地址存在. 对于高内存, 那个地址仅当这个页已被映射才存在. 这个函数在 <linux/mm.h> 中定义. 大部分情况下, 你想使用 kmap 的一个版本而不是 page_address.

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 为系统中的任何页返回一个内核虚拟地址. 对于低内存页, 它只返回页的逻辑地址; 对于高内存, kmap 在内核地址空间的一个专用部分中创建一个特殊的映射. 使用 kmap 创建的映射应当一直使用 kunmap 来释放;一个有限数目的这样的映射可用, 因此最好不要在它们上停留太长时间. kmap 调用维护一个计数器, 因此如果 2 个或 多个函数都在同一个页上调用 kmap, 正确的事情发生了. 还要注意 kmap 可能睡眠当没有映射可用时.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic 是 kmap 的一种高性能形式. 每个体系都给原子的 kmaps 维护一小列插口(专用的页表项); 一个 kmap_atomic 的调用者必须在 type 参数中告知系统使用这些插口中的哪个. 对驱动有意义的唯一插口是 KM_USER0 和 KM_USER1 (对于直接从来自用户空间的调用运行的代码), 以及 KM_IRQ0 和 KM_IRQ1(对于中断处理). 注意原子的 kmaps 必须被原子地处理; 你的代码不能在持有一个时睡眠. 还要注意内核中没有什么可以阻止 2 个函数试图使用同一个插口并且相互干扰

(尽管每个 CPU 有独特的一套插口). 实际上, 对原子的 kmap 插口的竞争看来不是个问题.

在本章后面和后续章节中当我们进入例子代码时, 我们看到这些函数的一些使用,

15.1.5. 页表

在任何现代系统上, 处理器必须有一个机制来转换虚拟地址到它的对应物理地址. 这个机制被称为一个页表; 它本质上是一个多级树型结构数组, 包含了虚拟-到-物理的映射和几个关联的标志. Linux 内核维护一套页表即便在没有直接使用这样页表的体系上.

设备驱动通常可以做的许多操作能涉及操作页表. 幸运的是对于驱动作者, 2.6 内核已经去掉了任何直接使用页表的需要. 结果是, 我们不描述它们的任何细节; 好奇的读者可能想读一下 Understanding The Linux Kernel 来了解完整的内容, 作者是 Daniel P. Bovet 和 Marco Cesati (O' Reilly).

15.1.6. 虚拟内存区

虚拟内存区(VMA)用来管理一个进程的地址空间的独特区域的内核数据结构. 一个 VMA 代表一个进程的虚拟内存的一个同质区域: 一个有相同许可标志和被相同对象(如, 一个文件或者交换空间)支持的连续虚拟地址范围. 它松散地对应于一个"段"的概念, 尽管可以更好地描述为"一个有它自己特性的内存对象". 一个进程的内存映射有下列区组成:

- 给程序的可执行代码(常常称为 text)的一个区.
- 给数据的多个区, 包括初始化的数据(它有一个明确的被分配的值, 在执行开始), 未初始化数据(BBS), [48] 以及程序堆栈.
- 给每个激活的内存映射的一个区域.

一个进程的内存区可看到通过 /proc/<pid/maps>(这里 pid, 当然, 用一个进程的 ID 来替换). /proc/self 是一个 /proc/id 的特殊情况, 因为它常常指当前进程. 作为一个例子, 这里是几个内存映射(我们添加了简短注释)

```
# cat /proc/1/maps look at init
08048000-0804e000 r-xp 00000000 03:01 64652
0804e000-0804f000 rw-p 00006000 03:01 64652
0804f000-08053000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 96278
40015000-40016000 rw-p 00014000 03:01 96278
40016000-40017000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:01 80290
4212e000-42131000 rw-p 0012e000 03:01 80290
42131000-42133000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0

/sbin/init text /sbin/init data zero-mapped BSS /lib/ld-2.3.2.so text /lib/ld-2.3.2.so
data BSS for ld.so /lib/tls/libc-2.3.2.so text /lib/tls/libc-2.3.2.so data BSS for libc
Stack segment vsyscall page
# rsh wolf cat /proc/self/maps #### x86-64 (trimmed)
```

```
00400000-00405000 r-xp 00000000 03:01 1596291 /bin/cat text
00504000-00505000 rw-p 00004000 03:01 1596291 /bin/cat data
00505000-00526000 rwxp 00505000 00:00 0 bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0 stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0 vsyscall
```

每行的字段是:

```
start-end perm offset major:minor inode image
```

每个在 /proc/*/maps (出来映象的名子) 对应 struct vm_area_struct 中的一个成员:

start end

　　这个内存区的开始和结束虚拟地址.

perm

　　带有内存区的读,写和执行许可的位掩码. 这个成员描述进程可以对属于这个区的页做什么. 成员的最后一个字符要么是给"私有"的 p 要么是给"共享"的 s.

offset

　　内存区在它被映射到的文件中的起始位置. 0 偏移意味着内存区开始对应文件的开始.

major minor

　　持有已被映射文件的设备的主次编号. 易混淆地, 对于设备映射, 主次编号指的是持有被用户打开的设备特殊文件的磁盘分区, 不是设备自身.

inode

　　被映射文件的 inode 号.

image

　　已被映射的文件名((常常在一个可执行映象中).

15.1.6.1. vm_area_struct 结构

当一个用户空间进程调用 mmap 来映射设备内存到它的地址空间, 系统通过一个新 VMA 代表那个映射来响应. 一个支持 mmap 的驱动(并且, 因此, 实现 mmap 方法)需要来帮助那个进程来完成那个 VMA 的初始化. 驱动编写者应当, 因此, 为支持 mmap 应至少有对 VMA 的最少的理解.

让我们看再 struct vm_area_struct 中最重要的成员(在 <linux/mm.h> 中定义). 这些成员应当被设备驱动在它们的 mmap 实现中使用. 注意内核维护 VMA 的链表和树来优化区查找, 并且 vm_area_struct 的几个成员被用来维护这个组织. 因此, VMA 不是有一个驱动任意创建的, 否则这个结构破坏了. VMA 的主要成员是下面(注意在这些成员和我们刚看到的 /proc 输出之间的相似)

```
unsigned long vm_start;
unsigned long vm_end;
```

被这个 VMA 覆盖的虚拟地址范围. 这些成员是在 /proc/*/maps 中出现的头 2 个字段.

```
struct file *vm_file;
```

一个指向和这个区(如果有一个)关联的 struct file 结构的指针.

```
unsigned long vm_pgoff;
```

文件中区的偏移, 以页计. 当一个文件和设备被映射, 这是映射在这个区的第一页的文件位置.

```
unsigned long vm_flags;
```

描述这个区的一套标志. 对设备驱动编写者最感兴趣的标志是 VM_IO 和 VM_RESERVUED. VM_IO 标志一个 VMA 作为内存映射的 I/O 区. 在其他方面, VM_IO 标志阻止这个区被包含在进程核转储中. VM_RESERVED 告知内存管理系统不要试图交换出这个 VMA; 它应当在大部分设备映射中设置.

```
struct vm_operations_struct *vm_ops;
```

一套函数, 内核可能会调用来在这个内存区上操作. 它的存在指示内存区是一个内核"对象", 象我们已经在全书中使用的 struct file.

```
void *vm_private_data;
```

驱动可以用来存储它的自身信息的成员.

象 struct vm_area_struct, vm_operations_struct 定义于 <linux/mm.h>; 它包括下面列出的操作. 这些操作是唯一需要来处理进程的内存需要的, 它们以被声明的顺序列出. 本章后面, 一些这些函数被实现.

```
void (*open)(struct vm_area_struct *vma);
```

open 方法被内核调用来允许实现 VMA 的子系统来初始化这个区. 这个方法被调用在任何时候有一个新的引用这个 VMA(当生成一个新进程, 例如). 一个例外是当这个 VMA 第一次被 mmap 创建时; 在这个情况下, 驱动的 mmap 方法被调用来替代.

void (*close)(struct vm_area_struct *vma);

当一个区被销毁, 内核调用它的关闭操作. 注意没有使用计数关联到 VMA; 这个区只被使用它的每个进程打开和关闭一次.

struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);

当一个进程试图存取使用一个有效 VMA 的页, 但是它当前不在内存中, nopage 方法被调用(如果它被定义)给相关的区. 这个方法返回 struct page 指针给物理页, 也许在从第 2 级存储中读取它之后. 如果 nopage 方法没有为这个区定义, 一个空页由内核分配.

int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);

这个方法允许内核"预错"页到内存, 在它们被用户空间存取之前. 对于驱动通常没有必要来实现这个填充方法.

15.1.7. 进程内存映射

内存管理难题的最后部分是进程内存映射结构, 它保持所有其他数据结构在一起. 每个系统中的进程(除了几个内核空间帮助线程)有一个 struct mm_struct (定义在 <linux/sched.h>), 它含有进程的虚拟内存区列表, 页表, 和各种其他的内存管理管理信息, 包括一个旗标(mmap_sem)和一个自旋锁(page_table_lock). 这个结构的指针在任务结构中; 在很少的驱动需要存取它的情况下, 通常的方法是使用 current->mm. 注意内存关联结构可在进程之间共享; Linux 线程的实现以这种方式工作, 例如.

这总结了我们对 Linux 内存管理数据结构的总体. 有了这些, 我们现在可以继续 mmap 系统调用的实现.

---

[47] 许多非-x86 体系可以有效工作在没有这里描述的内核/用户空间的划分, 因此它们可以在 32-位系统使用直到 4-GB 内核地址空间. 但是, 本节描述的限制仍然适用这样的系统当安装有多于 4GB 内存时.

[48] BSS 的名子是来自一个老的汇编操作符的历史遗物, 意思是"由符号开始的块". 可执行文件的 BSS 段不存储在磁盘上, 并且内核映射零页到 BSS 地址范围.

15.2. mmap 设备操作

内存映射是现代 Unix 系统最有趣的特性之一. 至于驱动, 内存映射可被实现来提供用户程序对设备内存的直接存取.

一个 mmap 用法的明确的例子可由查看给 X Windows 系统服务器的虚拟内存区的一个子集来见到:

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652 /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652 /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927 /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927 /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652 /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652 /dev/mem
...
```

X 服务器的 VMA 的完整列表很长, 但是大部分此处不感兴趣. 我们确实见到, 但是, /dev/mm 的 4 个不同映射, 它给出一些关于 X 服务器如何使用视频卡的内幕. 第一个映射在 a0000, 它是视频内存的在 640-KB ISA 孔里的标准位置. 再往下, 我们见到了大映射在 e8000000, 这个地址在系统中最高的 RAM 地址之上. 这是一个在适配器上的视频内存的直接映射.

这些区也可在 /proc/iomem 中见到:

```
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01

 e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI Bus #01

 fcc00000-fcc0ffff : 0000:01:00.0
```

映射一个设备意味着关联一些用户空间地址到设备内存. 无论何时程序在给定范围内读或写, 它实际上是在存取设备. 在 X 服务器例子里, 使用 mmap 允许快速和容易地存取视频卡内存. 对于一个象这样的性能关键的应用, 直接存取有很大不同.

如你可能期望的, 不是每个设备都出借自己给 mmap 抽象; 这样没有意义, 例如, 对串口或其他面向流的设备. mmap 的另一个限制是映射粒度是 PAGE_SIZE. 内核可以管理虚拟地址只在页表一级; 因此, 被映射区必须是 PAGE_SIZE 的整数倍并且必须位于是 PAGE_SIZE 整数倍开始的物理地址. 内核强制 size 的粒度通过做一个稍微大些的区域, 如果它的大小不是页大小的整数倍.

这些限制对驱动不是大的限制, 因为存取设备的程序是设备依赖的. 因为程序必须知道设备如何工作的, 程序员不会太烦于需要知道如页对齐这样的细节. 一个更大的限制存在当 ISA 设备被用在非 x86 平台时, 因为它们的 ISA 硬件视图可能不连续. 例如, 一些 Alpha 计算机将 ISA 内存看作一个分散的 8 位, 16 位, 32 位项的集合, 没有直接映射. 这种情况下, 你根本无法使用 mmap. 对不能进行直接映射 ISA 地址到 Alph 地址可能只发生在 32-位 和 64-位内存存取, ISA 可只做 8-位 和 16-位 发送, 并且没有办法来透明映射一个协议到另一个.

使用 mmap 有相当地优势当这样做可行的时候. 例如, 我们已经看到 X 服务器, 它传送大量数据到和从视频内存; 动态映射图形显示到用户空间提高了吞吐量, 如同一个 lseek/write 实现相反. 另一个典型例子是一个控制一个 PCI 设备的程序. 大部分 PCI

外设映射它们的控制寄存器到一个内存地址, 并且一个高性能应用程序可能首选对寄存器的直接存取来代替反复地调用 ioctl 来完成它的工作.

mmap 方法是 file_operation 结构的一部分, 当发出 mmap 系统调用时被引用. 用了 mmap, 内核进行大量工作在调用实际的方法之前, 并且, 因此, 方法的原型非常不同于系统调用的原型. 这不象 ioctl 和 poll 等调用, 内核不会在调用这些方法之前做太多.

系统调用如下一样被声明(如在 mmap(2) 手册页中描述的);

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

另一方面, 文件操作声明如下:

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

方法中的 filp 参数象在第 3 章介绍的那样, 而 vma 包含关于用来存取设备的虚拟地址范围的信息. 因此, 大量工作被内核完成; 为实现 mmap, 驱动只要建立合适的页表给这个地址范围, 并且, 如果需要, 用新的操作集合替换 vma->vm_ops.

有 2 个建立页表的方法:调用 remap_pfn_range 一次完成全部, 或者一次一页通过 nopage VMA 方法. 每个方法有它的优点和限制. 我们从"一次全部"方法开始, 它更简单. 从这里, 我们增加一个真实世界中的实现需要的复杂性.

15.2.1. 使用 remap_pfn_range

建立新页来映射物理地址的工作由 remap_pfn_range 和 io_remap_page_range 来处理, 它们有下面的原型:

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long phys_addr, unsigned long size, pgprot_t prot);
```

由这个函数返回的值常常是 0 或者一个负的错误值. 让我们看看这些函数参数的确切含义:

vma

页范围被映射到的虚拟内存区

virt_addr

重新映射应当开始的用户虚拟地址. 这个函数建立页表为这个虚拟地址范围从 virt_addr 到 virt_addr_size.

pfn

页帧号，对应虚拟地址应当被映射的物理地址。这个页帧号简单地是物理地址右移 PAGE_SHIFT 位。对大部分使用，VMA 结构的 vm_paoff 成员正好包含你需要的值。这个函数影响物理地址从 (pfn<<PAGE_SHIFT) 到 (pfn<<PAGE_SHIFT)+size.

size

正在被重新映射的区的大小，以字节.

prot

给新 VMA 要求的"protection". 驱动可(并且应当)使用在 vma->vm_page_prot 中找到的值.

给 remap_fpn_range 的参数是相当直接的，并且它们大部分是已经在 VMA 中提供给你，当你的 mmap 方法被调用时. 你可能好奇为什么有 2 个函数，但是. 第一个 (remap_pfn_range)意图用在 pfn 指向实际的系统 RAM 的情况下，而 io_remap_page_range 应当用在 phys_addr 指向 I/O 内存时. 实际上，这 2 个函数在每个体系上是一致的，除了 SPARC，并且你在大部分情况下被使用看到 remap_pfn_range . 为编写可移植的驱动，但是，你应当使用 remap_pfn_range 的适合你的特殊情况的变体.

另一种复杂性不得不处理缓存：常常地，引用设备内存不应当被处理器缓存. 常常系统 BIOS 做了正确设置，但是它也可能通过保护字段关闭特定 VMA 的缓存. 不幸的是，在这个级别上关闭缓存是高度处理器依赖的. 好奇的读者想看看来自 drivers/char/mem.c 的 pgprot_noncached 函数来找到包含什么. 我们这里不进一步讨论这个主题.

15.2.2. 一个简单的实现

如果你的驱动需要做一个简单的线性的设备内存映射，到一个用户地址空间，remap_pfn_range 几乎是所有你做这个工作真正需要做的. 下列的代码从 drivers/char/mem.c 中得来，并且显示了这个任务如何在一个称为 simple (Simple Implementation Mapping Pages with Little Enthusiasm)的典型模块中进行的.

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
 if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
 vma->vm_end - vma->vm_start,
 vma->vm_page_prot))
 return -EAGAIN;
 vma->vm_ops = &simple_remap_vm_ops;
 simple_vma_open(vma);
 return 0;
}
```

如你所见，重新映射内存只不过是调用 remap_pfn_rage 来创建必要的页表.

15.2.3. 添加 VMA 的操作

如我们所见，vm_area_struct 结构包含一套操作可以用到 VMA. 现在我们看看以一个简单的方式提供这些操作. 特别地，我们为 VMA 提供 open 和 close 操作. 这些操作被调用无论何时一个进程打开或关闭 VMA；特别地，open 方法被调用任何时候一个进程产生和创建一个对 VMA 的新引用. open 和 close VMA 方法被调用加上内核进行的处理，因此它们不需要重新实现任何那里完成的工作. 它们对于驱动存在作为一个方法来做任何它们可能要求的附加处理.

如同它所证明的，一个简单的驱动例如 simple 不需要做任何额外的特殊处理. 我们已创建了 open 和 close 方法，它打印一个信息到系统日志来通知大家它们已被调用. 不是特别有用，但是它确实允许我们来显示这些方法如何被提供，并且见到当它们被调用时.

到此，我们忽略了缺省的 vma->vm_ops 使用调用 printk 的操作:

```
void simple_vma_open(struct vm_area_struct *vma)
{
        printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n", vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
 printk(KERN_NOTICE "Simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops = {
 .open = simple_vma_open,
 .close = simple_vma_close,
};
```

为使这些操作为一个特定的映射激活，有必要存储一个指向 simple_remap_um_ops 指针在相关 VMA 的 vm_ops 成员中. 这常常在 mmap 方法中完成. 如果你回看 simple_remap_mmap 例子，你见到这些代码行:

```
vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
```

注意对 simple_vma_open 的明确调用. 因为 open 方法不在初始化 mmap 时调用，我们必须明确调用它如果我们要它运行.

15.2.4. 使用 nopage 映射内存

尽管 remap_pfn_range 对许多人工作得不错，如果不是大部分人，驱动 mmap 的实现有时有点更大的灵活性是必要的. 在这样的情况下，一个使用 nopage VMA 方法的实现可被调用.

一种 nopage 方法有用的情况可由 mremap 系统调用引起，它被应用程序用来改变一个被映射区的绑定地址. 如它所发生的，当一个被映射的 VMA 被 mremap 改变时内核不直接通知驱动. 如果这个 VMA 的大小被缩减，内核可静静地刷出不需要的页，而不必告诉驱动. 相反，如果这个 VMA 被扩大，当映射必须为新页建立时，驱动最终通过对 nopage

的调用发现，因此没有必要进行特殊的通知. nopage 方法，因此，如果你想支持 mremap 系统调用必须实现. 这里，我们展示一个简单的 nopage 实现给 simple 设备.

nopage 方法，记住，有下列原型:

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);
```

当一个用户进程试图存取在一个不在内存中的 VMA 中的一个页，相关的 nopage 函数被调用. 地址参数包含导致出错的虚拟地址，向下圆整到页的开始. nopage 函数必须定位并返回用户需要的页的 struct page 指针. 这个函数必须也负责递增它通过调用 get_page 宏返回的页的使用计数.

```
 get_page(struct page *pageptr);
```

这一步是需要的来保持在被映射页的引用计数正确. 内核为每个页维护这个计数；当计数到 0，内核知道这个页可被放置在空闲列表了. 当一个 VMA 被去映射，内核递减使用计数给区中每个页. 如果你的驱动在添加一个页到区时不递增计数，使用计数过早地成为 0，系统的整体性被破坏了.

nopage 方法也应当存储错误类型在由 type 参数指向的位置 -- 但是只当那个参数不为 NULL. 在设备驱动中，类型的正确值将总是 VM_FAULT_MINOR.

如果你使用 nopage，当调用 mmap 常常很少有工作来做；我们的版本看来象这样:

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
 unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
 vma->vm_flags |= VM_IO;
 vma->vm_flags |= VM_RESERVED;

 vma->vm_ops = &simple_nopage_vm_ops;
 simple_vma_open(vma);
 return 0;

}
```

mmap 必须做的主要的事情是用我们自己的操作来替换缺省的(NULL)vm_ops 指针. nopage 方法接着进行一次重新映射一页并且返回它的 struct page 结构的地址. 因为我们这里只实现一个到物理内存的窗口，重新映射的步骤是简单的：我们只需要定位并返回一个指向 struct page 的指针给需要的地址. 我们的 nopage 方法看来如下:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma, unsigned long address, int
*type)
{
 struct page *pageptr;
 unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
 unsigned long physaddr = address - vma->vm_start + offset;
 unsigned long pageframe = physaddr >> PAGE_SHIFT;
```

```
 if (!pfn_valid(pageframe))
 return NOPAGE_SIGBUS;
 pageptr = pfn_to_page(pageframe);
 get_page(pageptr);
 if (type)

 *type = VM_FAULT_MINOR;
 return pageptr;
}
```

因为，再一次，在这里我们简单地映射主内存，nopage 函数只需要找到正确的 struct page 给出错地址并且递增它的引用计数. 因此，事件的请求序列是计算需要地物理地址，并且通过右移它 PAGE_SHIFT 位转换它为以页帧号. 因为用户空间可以给我们任何它喜欢的地址，我们必须确保我们有一个有效的页帧；pfn_valid 函数为我们做这些. 如果地址超范围，我们返回 NOPAGE_SIGBUS，它产生一个总线信号被递交给调用进程.

否则，pfn_to_page 获得必要的 struct page 指针；我们可递增它的引用计数(使用调用 get_page)并且返回它.

nopage 方法正常地返回一个指向 struct page 的指针. 如果，由于某些原因，一个正常的页不能返回(即，请求的地址超出驱动的内存区)，NOPAGE_SIGBUS 可被返回来指示错误；这是上的简单代码所做的. nopage 也可以返回 NOPAGE_OOM 来指示由于资源限制导致的失败.

注意，这个实现对 ISA 内存区起作用，但是对那些在 PCI 总线上的不行. PCI 内存被映射在最高的系统内存之上，并且在系统内存中没有这些地址的入口. 因为没有 struct page 来返回一个指向的指针，nopage 不能在这些情况下使用；你必须使用 remap_pfn_range 代替.

如果 nopage 方法被留置为 NULL，处理页出错的内核代码映射零页到出错的虚拟地址. 零页是一个写时拷贝的页，它读作为 0，并且被用来，例如，映射 BSS 段. 任何引用零页的进程都看到：一个填满 0 的页. 如果进程写到这个页，它最终修改一个私有页. 因此，如果一个进程扩展一个映射的页通过调用 mremap，并且驱动还没有实现 nopage，进程结束以零填充的内存代替一个段错误.

15.2.5. 重新映射特定 I/O 区

所有的我们至今所见的例子是 /dev/mem 的重新实现；它们重新映射物理地址到用户空间. 典型的驱动，但是，想只映射应用到它的外设设备的小的地址范围，不是全部内存. 为了映射到用户空间只一个整个内存范围的子集，驱动只需要使用偏移. 下面为一个驱动做这个技巧来映射一个 simple_region_size 字节的区域，在物理地址 simple_region_start(应当是页对齐的) 开始：

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;
```

```
if (vsize > psize)
 return -EINVAL; /* spans too high */
remap_pfn_range(vma, vma_>vm_start, physical, vsize, vma->vm_page_prot);
```

除了计算偏移，这个代码引入了一个检查来报告一个错误当程序试图映射超过在目标设备的 I/O 区可用的内存. 在这个代码中，psize 是已指定了偏移后剩下的物理 I/O 大小，并且 vsize 是虚拟内存请求的大小；这个函数拒绝映射超出允许的内存范围的地址.

注意，用户空间可一直使用 mremap 来扩展它的映射，可能超过物理设备区的结尾. 如果你的驱动不能定义一个 nopage method，它从不会得到这个扩展的通知，并且额外的区映射到零页. 作为一个驱动编写者，你可能很想阻止这种行为；映射理由到你的区的结尾不是一个明显的坏事情，但是很不可能程序员希望它发生.

最简单的方法来阻止映射扩展是实现一个简单的 nopage 方法，它一直导致一个总线信号被发送给出错进程. 这样的一个方法可能看来如此：

```
struct page *simple_nopage(struct vm_area_struct *vma,
 unsigned long address, int *type);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */}
```

如我们已见到的，nopage 方法只当进程解引用一个地址时被调用，这个地址在一个已知的 VMA 中但是当前没有有效的页表入口给这个 VMA. 如果有已使用 remap_pfn_range 来映射全部设备区，这里展示的 nopage 方法只被调用来引用那个区外部. 因此，它能够安全地返回 NOPAGE_SIGBUS 来指示一个错误. 当然，一个更加完整的 nopage 实现可以检查是否出错地址在设备区内，并且如果是这样进行重新映射. 但是，再一次，nopage 无法在 PCI 内存区工作，因此 PCI 映射的扩展是不可能的.

15.2.6. 重新映射 RAM

remap_pfn_range 的一个有趣的限制是它只存取保留页和在物理内存顶之上的物理地址. 在 Linux，一个物理地址页被标志为"保留的"在内存映射中来指示它对内存管理是不可用的. 在 PC 上，例如，640 KB 和 1MB 之间被标记为保留的，如同驻留内核代码自身的页. 保留页被锁定在内存并且是唯一可被安全映射到用户空间的；这个限制是系统稳定的一个基本要求.

因此，remap_pfn_range 不允许你重新映射传统地址，这包括你通过调用 get_free_page 获得的. 相反，它映射在零页. 所有都看来正常，除了进程见到私有的，零填充的页而不是它在期望的被重新映射的 RAM. 这个函数做了大部分硬件驱动需要来做的所有事情，因为它能够重新映射高端 PCI 缓冲和 ISA 内存.

remap_pfn_range 的限制可通过运行 mapper 见到，其中一个例子程序在 misc-progs 在 O'Reilly 的 FTP 网站提供的文件. mapper 是一个简单的工具可用来快速测试 mmap 系统调用；它映射由命令行选项指定的一个文件的只读部分，并且输出被映射的区到标准输出. 下面的部分，例如，显示 /dev/mem 没有映射位于地址 64 KB 的物理页 --相反，我们看到一个页充满 0 (例子中的主机是一台 PC，但是结果应该在其他平台上相同).

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
```

```
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

remap_pfn_range 处理 RAM 的不能之处说明基于内存的设备如 scull 不能轻易实现 mmap, 因为它的设备内存是传统内存, 不是 I/O 内存. 幸运的是, 一个相对容易的方法对任何需要映射 RAM 到用户空间的驱动都可用; 它使用我们前面已见过的 nopage 方法.

15.2.6.1. 使用 nopage 方法重新映射 RAM

映射真实内存到用户空间的方法是使用 vm_ops-<nopage 来一次一个地处理页错. 一个简单的实现是 scullp 模块的一部分, 在第 8 张介绍.

scullp 是一个面向页的字符设备. 因为它是面向页的, 它可以在它的内存上实现 mmap. 实现内存映射的代码使用一些在"Linux 中的内存管理"一节中介绍的概念.

在检查代码前, 让我们查看影响在 scullp 中的 mmap 实现的设计选择.

- scullp 只要设备被映射就不会释放设备内存. 这是策略问题而非一个需求, 并且它不同于 scull 和类似设备的行为, 它们被截短为 0 当为写而打开时. 对释放一个映射的 scullp 设备的拒绝, 允许一个进程覆盖被其他进程映射的区., 因此你可以测试并且看进程和设备内存如何交互. 为避免释放一个映射设备, 驱动必须保持一个激活映射的计数; 在设备结构中的 vmas 成员被用来作此目的.
- 内存映射仅当 scullp 的 order 参数(在模块加载时间设置)是 0 时进行. 这个参数控制 __get_free_pages 如何被调用(见第 8 章"get_free_page 及其友" 一节). 0 order 的限制(这强制一次分配一页, 而不是以大的组)被 __get_free_pages 的内部所规定, 它是 scullp 所使用的分配函数. 为最大化分配性能, Linux 内核维护一个空闲页列表给每个分配级别, 并且只有在一个簇中第一页的引用计数被 get_free_pages 递增以及被 free_pages 递减. mmap 方法对一个 scullp 设备被禁止, 如果分配级大于 0, 因为 nopage 处理单个页而不是一簇页. scullp 不知道如何正确管理是高级别分配的一部分的页的引用计数.(如果你需要重新回顾 scullp 和 内存分配级别值, 返回第 8 章的"一个使用整页的 scull: scullp"一节.)

0-级的限制大部分是用来保持代码简单. 它可能正确实现 mmap 给多页分配, 通过使用页的使用计数, 但是它可能只增加了例子的复杂性而没有介绍任何有趣的信息.

打算根据刚刚概括的规则来映射 RAM 的代码, 需要实现 open, close, 和 nopage VMA 方法; 它还需要存取内存映射来调整页使用计数.

这个 scullp_mmap 的实现非常短, 因为它依赖 nopage 函数来做所有的感兴趣的工作:

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{

 struct inode *inode = filp->f_dentry->d_inode;
 /* refuse to map if order is not 0 */
 if (scullp_devices[iminor(inode)].order)
```

```
 return -ENODEV;

 /* don't do anything here: "nopage" will fill the holes */
 vma->vm_ops = &scullp_vm_ops;
 vma->vm_flags |= VM_RESERVED;
 vma->vm_private_data = filp->private_data;
 scullp_vma_open(vma);
 return 0;
}
```

if 语句的目的是避免映射分配级别不是 0 的设备. scullp 的操作存储在 vm_ops 成员，并且一个指向设备结构的指针藏于 vm_private_data 成员. 最后，vm_ops->open 被调用来更新设备的激活映射的计数.

open 和 close 简单地跟踪映射计数并如下定义:

```
void scullp_vma_open(struct vm_area_struct *vma)
{
 struct scullp_dev *dev = vma->vm_private_data;
 dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
 struct scullp_dev *dev = vma->vm_private_data;
 dev->vmas--;
}
```

大部分地工作接下来由 nopage 进行. 在 scullp 实现中，给 nopage 的地址参数被用来计算设备中的偏移; 这个偏移接着被用来在 scullp 内存树中查找正确的页.

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma, unsigned long address, int
*type)
{
        unsigned long offset;
        struct scullp_dev *ptr, *dev = vma->vm_private_data;
        struct page *page = NOPAGE_SIGBUS;
        void *pageptr = NULL; /* default to "missing" */

        down(&dev->sem);
        offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
        if (offset >= dev->size)
                goto out; /* out of range */

        /*
        * Now retrieve the scullp device from the list,then the page.
        * If the device has holes, the process receives a SIGBUS when
        * accessing the hole.
        */
        offset >>= PAGE_SHIFT; /* offset is a number of pages */
        for (ptr = dev; ptr && offset >= dev->qset;)
        {
```

```
                ptr = ptr->next;
                offset -= dev->qset;
        }
        if (ptr && ptr->data)
                pageptr = ptr->data[offset];
        if (!pageptr)
                goto out; /* hole or end-of-file */
        page = virt_to_page(pageptr);

        /* got it, now increment the count */
        get_page(page);
        if (type)
                *type = VM_FAULT_MINOR;
out:
        up(&dev->sem);
        return page;
}
```

scullp 使用由 get_free_pages 获取的内存. 那个内存使用逻辑地址寻址, 因此所有的 scullp_nopage 为获得一个 struct page 指针不得不做的是调用 virt_to_page.

现在 scullp 设备如同期望般工作了, 就象你在这个从 mapper 工具中的例子输出能见到的. 这里, 我们发送一个 /dev 的目录列表(一个长的)到 scullp 设备并且接着使用 mapper 工具来查看这个列表的各个部分连同 mmap.

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw-------1 root root 10, 10 Sep 15 07:40 adbmouse
crw-r--r--1 root root 10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200 mapped "/dev/scullp" from 8192 (0x00002000) to 8392
(0x000020c8)
d0h1494
brw-rw---- 1 root  floppy  2,  92 Sep 15 07:40 fd0h1660
brw-rw---- 1 root  floppy  2,  20 Sep 15 07:40 fd0h360
brw-rw---- 1 root  floppy  2,  12 Sep 15 07:40 fd0H360
```

15.2.7. 重映射内核虚拟地址

尽管它极少需要, 看一个驱动如何使用 mmap 映射一个内核虚拟地址到用户空间是有趣的. 记住, 一个真正的内核虚拟地址, 是一个由诸如 vmalloc 的函数返回的地址 -- 就是, 一个映射到内核页表中的虚拟地址. 本节的代码来自 scullv, 这是如同 scullp 但是通过 vmalloc 分配它的存储的模块.

大部分的 scullv 实现如同我们刚刚见到的 scullp, 除了没有必要检查控制内存分配的 order 参数. 这个的原因是 vmalloc 分配它的页一次一个, 因为单页分配比多页分配更加可能成功. 因此, 分配级别问题不适用 vmalloc 分配的空间.

此外, 在由 scullp 和 scullv 使用的 nopage 实现中只有一个不同. 记住, scullp 一旦它发现感兴趣的页, 将使用 virt_to_page 来获得对应的 struct page 指针. 那个函

数不使用内核虚拟地址，但是．相反，你必须使用 mvalloc_to_page．因此 scullv 版本的 nopage 的最后部分看来如此：

```
/*
* After scullv lookup, "page" is now the address of the page
* needed by the current process. Since it's a vmalloc address,
* turn it into a struct page.
*/
page = vmalloc_to_page(pageptr);

/* got it, now increment the count */
get_page(page);
if (type)
        *type = VM_FAULT_MINOR;
out:
up(&dev->sem);
return page;
```

基于这个讨论，你可能也想映射由 ioremap 返回的地址到用户空间．但是，那可能是一个错误；来自 ioremap 的地址是特殊的并且不能作为正常的内核虚拟地址对待．相反，你应当使用 remap_pfn_range 来重新映射 I/O 内存区到用户空间．

15.3. 进行直接 I/O

大部分 I/O 操作是通过内核缓冲的．一个内核空间缓冲区的使用允许一定程度的用户空间和实际设备的分离；这种分离能够使编程容易并且还可以在许多情况下有性能的好处．但是，有这样的情况它对于进行 I/O 直接到或从一个用户空间缓冲区是有好处的．如果正被传输的数据量大，不使用一个额外的拷贝直接通过内核空间传输数据可以加快事情进展．

2.6 内核中一个直接 I/O 使用的例子是 SCSI 磁带驱动．流动的磁带能够传送大量数据通过系统，并且磁带传送常常是面向记录的，因此在内核中缓冲数据没有好处．因此，当条件正确(用户空间缓冲区是页对齐的，例如)，SCSI 磁带驱动进行它的 I/O 而不拷贝数据．

就是说，重要的是认识到直接 I/O 不是一直提供人们期望的性能提高．设置直接 I/O (它调用出错换入并且除下相关的用户空间)的开销可能是不小的，并且被缓冲的 I/O 的好处丢失了．例如，直接 I/O 的使用要求 write 系统调用同步操作；否则应用程序不能知道什么时间它可以重新使用它的 I/O 缓冲．停止应用程序直到每个 write 完成可能拖慢事情，这是为什么使用直接 I/O 的应用程序也常常使用异步 I/O 操作的原因．

事情的真正内涵是，在任何情况下，在一个字符驱动实现直接 I/O 常常是不必要并且可能是有害的．你应当只在你确定缓冲的 I/O 的开销确实拖慢了系统的情况下采取这个步骤．还要注意，块和网络驱动不必担心实现直接 I/O；这 2 种情况下，内核中的高级的代码在需要时建立和使用直接 I/O，并且驱动级别的代码甚至不需要知道直接 I/O 在被进行中．

实现直接 I/O 的关键是一个称为 get_user_pages 的函数, 它在 <linux/mm.h> 中定义使用下列原型:

```
int get_user_pages(struct task_struct *tsk,
 struct mm_struct *mm,
 unsigned long start,
 int len,
 int write,
 int force,
 struct page **pages,
 struct vm_area_struct **vmas);
```

这个函数有几个参数:

tsk

一个指向进行 I/O 的任务的指针; 它的主要目的是告知内核谁应当负责任何一个当设置缓冲时导致的页错. 这个参数几乎一直作为 current 传递.

mm

一个内存管理结构的指针, 描述被映射的地址空间. mm_struct 结构是捆绑一个进程的虚拟地址空间所有的部分在一起的. 对于驱动的使用, 这个参数应当一直是 current->mm.

start len

start 是(页对齐的)用户空间缓冲的地址, 并且 len 是缓冲的长度以页计.

write force

如果 write 是非零, 这些页被映射来写(当然, 隐含着用户空间在进行一个读操作). force 标志告知 get_user_pages 来覆盖在给定页上的保护, 来提供要求的权限; 驱动应当一直传递 0 在这里.

pages vmas

输出参数. 在成功完成后, 页包含一系列指向 struct page 结构的指针来描述用户空间缓冲, 并且 vmas 包含指向被关联的 VMA 的指针. 这些参数应当, 显然, 指向能够持有至少 len 个指针的数组. 任一个参数可能是 NULL, 但是你需要, 至少, struct page 指针来实际对缓冲操作.

get_user_pages 是一个低级内存管理函数, 带一个相称的复杂的接口. 它还要求给这个地址空间的 mmap 读者/写者 旗标在调用前被以读模式获得. 结果是, 对 get_user_pages 常常看来象:

```
down_read(&current->mm->mmap_sem);
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

返回值是实际映射的页数，它可能小于请求的数目(但是大于 0).

一旦成功完成，调用者有一个页数组指向用户空间缓冲，它被锁入内存. 为直接在缓冲上操作，内核空间代码必须将每个 struct page 指针转换为一个内核虚拟地址，使用 kmap 或者 kmap_atomic. 常常地，但是，对于可以使用直接 I/O 的设备在使用 DMA 操作，因此你的驱动将可能想从 struct page 指针数组创建一个发散/汇聚列表. 我们在 "发散/汇聚映射"一节中讨论如何做这个.

一旦你的直接 I/O 操作完成了，你必须释放用户页. 在这样做之前，但是，你必须通知内核如果你改变了这些页的内容. 否则，内核可能认为这些页是"干净"的，意味着它们匹配一个在交换设备中发现的一个拷贝，并且释放它们不写出它们到备份存储. 因此，如果你已改变了这些页(响应一个用户空间写请求)，你必须标志每个被影响到的页为脏，使用一个调用:

```
void SetPageDirty(struct page *page);
```

(这个宏定义在 <linux/page-flags.h>). 进行这个操作的代码首先检查来保证页不在内存映射的保留部分，这部分从不被换出. 因此，代码常常看来如此:

```
if (! PageReserved(page))
 SetPageDirty(page);
```

因为用户空间内存正常地不置为保留的，这个检查严格地不应当是必要的，但是当你深入内存管理子系统时，最好全面并且仔细.

不管这些页是否已被改变，它们必须从页缓存中释放，或者它们一直留在那里. 这个调用是:

```
void page_cache_release(struct page *page);
```

这个调用应当，当然，在页已被标识为脏之后进行，如果需要.

15.3.1. 异步 I/O

增加到 2.6 内核的一个新的特性是异步 I/O 能力. 异步 I/O 允许用户空间来初始化操作而不必等待它们的完成; 因此，一个应用程序可以在它的 I/O 在进行中时做其他的处理. 一个复杂的，高性能的应用程序还可使用异步 I/O 来使多个操作在同一个时间进行.

异步 I/O 的实现是可选的，并且很少几个驱动作者关心; 大部分设备不会从这个能力中受益. 如同我们将在接下来的章节中见到的，块和网络驱动在整个时间是完全异步的，因此只有字符驱动对于明确的异步 I/O 支持是候选的. 一个字符设备能够从这个支持中受益，如果有好的理由来使多个 I/O 操作在任一给定时间同时进行. 一个好例子是流化磁带驱动，这里这个驱动可停止并且明显慢下来如果 I/O 操作没有尽快到达. 一个应用程序试图从一个流驱动中获得最好的性能，可以使用异步 I/O 来使多个操作在任何时间准备好进行.

对于少见的需要实现异步 I/O 的驱动作者, 我们提供一个快速的关于它如何工作的概观. 我们涉及异步 I/O 在本章, 因为它的实现几乎一直也包括直接 I/O 操作(如果你在内核中缓冲数据, 你可能常常实现异步动作而不必在用户空间出现不必要的复杂性).

支持异步 I/O 的驱动应当包含 <linux/aio.h>. 有 3 个 file_operation 方法给异步 I/O 实现:

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
 size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
 size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

aio_fsync 操作只对文件系统代码感兴趣, 因此我们在此不必讨论它. 其他 2 个, aio_read 和 aio_write, 看起来非常象常规的 read 和 write 方法, 但是有几个例外. 一个是 offset 参数由值传递; 异步操作从不改变文件位置, 因此没有理由传一个指针给它. 这些方法还使用 iocb ("I/O 控制块")参数, 这个我们一会儿就到.

aio_read 和 aio_write 方法的目的是初始化一个读或写操作, 在它们返回时可能完成或者可能没完成. 如果有可能立刻完成操作, 这个方法应当这样做并且返回通常的状态: 被传输的字节数或者一个负的错误码. 因此, 如果你的驱动有一个称为 my_read 的读方法, 下面的 aio_read 方法是全都正确的(尽管特别无意义):

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer, ssize_t count, loff_t offset)
{
 return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

注意, struct file 指针在 kocb 结构的 ki_filp 成员中.

如果你支持异步 I/O, 你必须知道这个事实, 内核可能, 偶尔, 创建"异步 IOCB". 它们是, 本质上, 必须实际上被同步执行的异步操作. 有人可能非常奇怪为什么要这样做, 但是最好只做内核要求做的. 同步操作在 IOCB 中标识; 你的驱动应当询问状态, 使用:

```
int is_sync_kiocb(struct kiocb *iocb);
```

如果这个函数返回一个非零值, 你的驱动必须同步执行这个操作.

但是, 最后, 所有这个结构的意义在于使能异步操作. 如果你的驱动能够初始化这个操作(或者, 简单地, 将它排队到它能够被执行时), 它必须做两件事情: 记住它需要知道的关于这个操作的所有东西, 并且返回 -EIOCBQUEUED 给调用者. 记住操作信息包括安排对用户空间缓冲的存取; 一旦你返回, 你将不再有机会来存取缓冲, 当再调用进程的上下文运行时. 通常, 那意味着你将可能不得不建立一个直接内核映射(使用 get_user_pages) 或者一个 DMA 映射. -EIOCBQUEUED 错误码指示操作还没有完成, 并且它最终的状态将之后传递.

当"之后"到来时, 你的驱动必须通知内核操作已经完成. 那通过调用 aio_complete 来完成:

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

这里, iocb 是起初传递给你的同一个 IOCB, 并且 res 是这个操作的通常的结果状态. res2 是将被返回给用户空间的第 2 个结果码; 大部分的异步 I/O 实现作为 0 传递 res2. 一旦你调用 aio_complete, 你不应当再碰 IOCB 或者用户缓冲.

15.3.1.1. 一个异步 I/O 例子

例子代码中的面向页的 scullp 驱动实现异步 I/O. 实现是简单的, 但是足够来展示异步操作应当如何被构造.

aio_read 和 aio_write 方法实际上不做太多:

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t pos)
{
        return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf, size_t count, loff_t
pos)
{
        return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

这些方法仅仅调用一个普通的函数:

```
struct async_work
{
        struct kiocb *iocb;
        int result;
        struct work_struct work;

};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf, size_t count, loff_t
pos)
{
        struct async_work *stuff;
        int result;

        /* Copy now while we can access the buffer */
        if (write)
                result = scullp_write(iocb->ki_filp, buf, count, &pos);
        else
                result = scullp_read(iocb->ki_filp, buf, count, &pos);

        /* If this is a synchronous IOCB, we return our status now. */
        if (is_sync_kiocb(iocb))
                return result;

        /* Otherwise defer the completion for a few milliseconds. */
        stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
        if (stuff == NULL)
```

```
                return result; /* No memory, just complete now */
        stuff->iocb = iocb;
        stuff->result = result;
        INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
        schedule_delayed_work(&stuff->work, HZ/100);
        return -EIOCBQUEUED;
}
```

一个更加完整的实现应当使用 get_user_pages 来映射用户缓冲到内核空间. 我们选择来使生活简单些, 通过只拷贝在 outset 的数据. 接着调用 is_sync_kiocb 来看是否这个操作必须同步完成; 如果是, 结果状态被返回, 并且我们完成了. 否则我们记住相关的信息在一个小结构, 通过一个工作队列来为"完成"而安排, 并且返回 -EIOCBQUEUED. 在这点上, 控制返回到用户空间.

之后, 工作队列执行我们的完成函数:

```
static void scullp_do_deferred_op(void *p)
{
 struct async_work *stuff = (struct async_work *) p;
 aio_complete(stuff->iocb, stuff->result, 0);
 kfree(stuff);
}
```

这里, 只是用我们保存的信息调用 aio_complete 的事情. 一个真正的驱动的异步 I/O 实现是有些复杂, 当然, 但是它遵循这类结构.

15.4. 直接内存存取

直接内存存取, 或者 DMA, 是结束我们的内存问题概览的高级主题. DMA 是硬件机制允许外设组件来直接传输它们的 I/O 数据到和从主内存, 而不需要包含系统处理器. 这种机制的使用能够很大提高吞吐量到和从一个设备, 因为大量的计算开销被削减了.

15.4.1. 一个 DMA 数据传输的概况

在介绍程序细节之前, 让我们回顾一个 DMA 传输如何发生的, 只考虑输入传输来简化讨论.

数据传输可由 2 种方法触发:或者软件请求数据(通过一个函数例如 read)或者硬件异步推数据到系统.

在第一种情况, 包含的步骤总结如下:

- 1. 当一个进程调用 read, 驱动方法分配一个 DMA 缓冲并引导硬件来传输它的数据到那个缓冲. 这个进程被置为睡眠.
- 2. 硬件写数据到这个 DMA 缓冲并且在它完成时引发一个中断.
- 3. 中断处理获得输入数据, 确认中断, 并且唤醒进程, 它现在可以读数据了.

第 2 种情况到来是当 DMA 被异步使用. 例如, 这发生在数据获取设备, 它在没有人读它们的时候也持续推入数据. 在这个情况下, 驱动应当维护一个缓冲以至于后续的读调用能返回所有的累积的数据给用户空间. 这类传输包含的步骤有点不同:

- 1. 硬件引发一个中断来宣告新数据已经到达.
- 2. 中断处理分配一个缓冲并且告知硬件在哪里传输数据.
- 3. 外设写数据到缓冲并且引发另一个中断当完成时.
- 处理者分派新数据, 唤醒任何相关的进程, 并且负责杂务.

异步方法的变体常常在网卡中见到. 这些卡常常期望见到一个在内存中和处理器共享的环形缓冲(常常被称为一个 DMA 的缓冲); 每个到来的报文被放置在环中下一个可用的缓冲, 并且发出一个中断. 驱动接着传递网络本文到内核其他部分并且在环中放置一个新 DMA 缓冲.

在所有这些情况中的处理的步骤都强调, 有效的 DMA 处理依赖中断报告. 虽然可能实现 DMA 使用一个轮询驱动, 它不可能有意义, 因为一个轮询驱动可能浪费 DMA 提供的性能益处超过更容易的处理器驱动的I/O.[49]

在这里介绍的另一个相关项是 DMA 缓冲. DMA 要求设备驱动来分配一个或多个特殊的适合 DMA 的缓冲. 注意许多驱动分配它们的缓冲在初始化时并且使用它们直到关闭 -- 在之前列表中的分配一词, 意思是"获得一个之前分配的缓冲".

15.4.2. 分配 DMA 缓冲

本节涵盖 DMA 缓冲在底层的分配; 我们稍后介绍一个高级接口, 但是来理解这里展示的内容仍是一个好主意.

随 DMA 缓冲带来的主要问题是, 当它们大于一页, 它们必须占据物理内存的连续页因为设备使用 ISA 或者 PCI 系统总线传输数据, 它们都使用物理地址. 注意有趣的是这个限制不适用 SBus (见 12 章的"SBus"一节), 它在外设总线上使用虚拟地址. 一些体系结构还可以在 PCI 总线上使用虚拟地址, 但是一个可移植的驱动不能依赖这个功能.

尽管 DMA 缓冲可被分配或者在系统启动时或者在运行时, 模块只可在运行时分配它们的缓冲. (第 8 章介绍这些技术; "获取大缓冲"一节涵盖在系统启动时分配, 而"kmalloc 的真实"和"get_free_page 和其友"描述在运行时分配). 驱动编写者必须关心分配正确的内存,当它被用做 DMA 操作时; 不是所有内存区是合适的. 特别的, 在一些系统中的一些设备上高端内存可能不为 DMA 工作 - 外设完全无法使用高端地址.

在现代总线上的大部分设备可以处理 32-位 地址, 意思是正常的内存分配对它们是刚刚好的. 一些 PCI 设备, 但是, 不能实现完整的 PCI 标准并且不能使用 32-位 地址. 并且 ISA 设备, 当然, 限制只在 24-位 地址.

对于有这种限制的设备, 内存应当从 DMA 区进行分配, 通过添加 GFP_DMA 标志到 kmalloc 或者 get_free_pages 调用. 当这个标志存在, 只有可用 24-位 寻址的内存被分配. 另一种选择, 你可以使用通用的 DMA 层(我们马上讨论这个)来分配缓冲以解决你的设备的限制.

15.4.2.1. 自己做分配

我们已见到 get_free_pages 如何分配直到几个 MByte (由于 order 可以直到 MAX_ORDER, 当前是 11), 但是高级数的请求容易失败当请求的缓冲远远小于 128 KB, 因为系统内存时间长了变得碎裂.[50]

当内核无法返回请求数量的内存或者当你需要多于 128 KB(例如, 一个通常的 PCI 帧抓取的请求), 一个替代返回 -ENOMEM 的做法是在启动时分配内存或者保留物理 RAM 的顶部给你的缓冲. 我们在第 8 章的 "获得大量缓冲" 一节描述在启动时间分配, 但是它对模块是不可用的. 保留 RAM 的顶部是通过在启动时传递一个 mem= 参数给内核实现的. 例如, 如果你有 256 MB, 参数 mem=255M 使内核不使用顶部的 MByte. 你的模块可能后来使用下列代码来获得对这个内存的存取:

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

分配器, 配合本书的例子代码的一部分, 提供了一个简单的 API 来探测和管理这样的保留 RAM 并且已在几个体系上被成功使用. 但是, 这个技巧当你有一个高内存系统时无效(即, 一个有比适合 CPU 地址空间更多的物理内存的系统).

当然, 另一个选项, 是使用 GFP_NOFAIL 来分配你的缓冲. 这个方法, 但是, 确实严重地对内存管理子系统有压力, 并且它冒锁住系统的风险; 最好是避免除非确实没有其他方法.

如果你分配一个大 DMA 缓冲到这样的长度, 但是, 值得想一下替代的方法. 如果你的设备可以做发散/汇聚 I/O, 你可以分配你的缓冲以更小的片段并且让设备做其他的. 发散/汇聚 I/O 也可以用当进行直接 I/O 到用户空间时, 它可能是最好地解决方法当需要一个真正大缓冲时.

15.4.3. 总线地址

一个使用 DMA 的设备驱动必须和连接到接口总线的硬件通讯, 总线使用物理地址, 而程序代码使用虚拟地址.

事实上, 情况比这个稍微有些复杂. 基于 DMA 的硬件使用总线地址, 而不是物理地址. 尽管 ISA 和 PCI 总线地址在 PC 上完全是物理地址, 这对每个平台却不总是真的. 有时接口总线被通过桥接电路连接, 它映射 I/O 地址到不同的物理地址. 一些系统甚至有一个页映射机制, 使任意的页连续出现在外设总线.

在最低级别(再次, 我们将马上查看一个高级解决方法), Linux 内核提供一个可移植的方法, 通过输出下列函数, 在 <asm/io.h> 定义. 这些函数的使用不被推荐, 因为它们只在有非常简单的 I/O 体系的系统上正常工作; 但是, 你可能遇到它们当使用内核代码时.

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

这些函数进行一个简单的转换在内核逻辑地址和总线地址之间. 它们在许多情况下不工作, 一个 I/O 内存管理单元必须被编程的地方或者必须使用反弹缓冲的地方. 做这个转换的正确方法是使用通用的 DMA 层, 因此我们现在转移到这个主题.

15.4.4. 通用 DMA 层

DMA 操作, 最后, 下到分配一个缓冲并且传递总线地址到你的设备. 但是, 编写在所有体系上安全并正确进行 DMA 的可移植启动的任务比想象的要难. 不同的系统有不同的概念, 关于缓存一致性应当如何工作的概念; 如果你不正确处理这个问题, 你的驱动可能破坏内存. 一些系统有复杂的总线硬件, 它使 DMA 任务更容易 - 或者更难. 并且不是所有的系统可以在内存所有部分进行 DMA. 幸运的是, 内核提供了一个总线和体系独立的 DMA 层来对驱动作者隐藏大部分这些问题. 我们非常鼓励你来使用这个层来 DMA 操作, 在任何你编写的驱动中.

下面的许多函数需要一个指向 struct device 的指针. 这个结构是 Linux 设备模型中设备的低级表示. 它不是驱动常常必须直接使用的东西, 但是你确实需要它当使用通用 DMA 层时. 常常地, 你可发现这个结构, 深埋在描述你的设备的总线. 例如, 它可在 struct pci_device 或者 struct usb_device 中发现它作为 dev 成员. 设备结构在 14 章中详细描述.

使用下面函数的驱动应当包含 <linux/dma-mapping.h>.

15.4.4.1. 处理困难硬件

在尝试 DMA 之前必须回答的第一个问题是给定设备是否能够在当前主机上做这样的操作. 许多设备受限于它们能够寻址的内存范围, 因为许多理由. 缺省地, 内核假定你的设备能够对任何 32-位 地址进行 DMA. 如果不是这样, 你应当通知内核这个事实, 使用一个调用:

```
int dma_set_mask(struct device *dev, u64 mask);
```

mask 应当显示你的设备能够寻址的位; 如果它被限制到 24 位, 例如, 你要传递 mask 作为 0x0FFFFFF. 返回值是非零如果使用给定的 mask 可以 DMA; 如果 dma_set_mask 返回 0, 你不能对这个设备使用 DMA 操作. 因此, 设备的驱动中的初始化代码限制到 24-位 DMA 操作可能看来如:

```
if (dma_set_mask (dev, 0xffffff))
        card->use_dma = 1;
else
{
        card->use_dma = 0; /* We'll have to live without DMA */
        printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

再次, 如果你的设备支持正常的, 32-位 DMA 操作, 没有必要调用 dma_set_mask.

15.4.4.2. DMA 映射

一个 DMA 映射是分配一个 DMA 缓冲和产生一个设备可以存取的地址的结合. 它试图使用一个简单的对 virt_to_bus 的调用来获得这个地址, 但是有充分的理由来避免那个方法.

它们中的第一个是合理的硬件带有一个 IOMMU 来为总线提供一套映射寄存器. IOMMU 可为任何物理内存安排来出现在设备可存取的地址范围内, 并且它可使物理上散布的缓冲对设备看来是连续的. 使用 IOMMU 需要使用通用的 DMA 层; virt_to_bus 不负责这个任务.

注意不是所有的体系都有一个 IOMMU; 特别的, 流行的 x86 平台没有 IOMMU 支持. 一个正确编写的驱动不需要知道它在之上运行的 I/O 支持硬件, 但是.

为设备设置一个有用的地址可能也, 在某些情况下, 要求一个反弹缓冲的建立. 反弹缓冲是当一个驱动试图在一个外设不能达到的地址上进行 DMA 时创建的, 比如一个高内存地址. 数据接着根据需要被拷贝到和从反弹缓冲. 无需说, 反弹缓冲的使用能拖慢事情, 但是有时没有其他选择.

DMA 映射也必须解决缓存一致性问题. 记住现代处理器保持最近存取的内存区的拷贝在一个快速的本地缓冲中; 如果没有这个缓存, 合理的性能是不可能的. 如果你的设备改变主存一个区, 会强制使任何包含那个区的处理器缓存被失效; 负责处理器可能使用不正确的主存映象, 并且导致数据破坏. 类似地, 当你的设备使用 DMA 来从主存中读取数据, 任何对那个驻留在处理器缓存的内存的改变必须首先被刷新. 这些缓存一致性问题可以产生无头的模糊和难寻的错误, 如果编程者不小心. 一个体系在硬件中管理缓存一致性, 但是其他的要求软件支持. 通用的 DMA 层深入很多来保证在所有体系上事情都正确工作, 但是, 如同我们将见到的, 正确的行为要求符合一些规则.

DMA 映射设置一个新类型, dma_addr_t, 来代表总线地址. 类型 dma_addr_t 的变量应当被驱动当作不透明的; 唯一可允许的操作是传递它们到 DMA 支持过程和设备自身. 作为一个总线地址, dma_addr_t 可导致不期望的问题如果被 CPU 直接使用.

PCI 代码在 2 类 DMA 映射中明显不同, 依赖 DMA 缓冲被期望停留多长时间:

Coherent DMA mappings

连贯的 DMA 映射. 这些映射常常在驱动的生命期内存在. 一个连贯的缓冲必须是同时对 CPU 和外设可用(其他的映射类型, 如同我们之后将看到的, 在任何给定时间只对一个或另一个可用). 结果, 一致的映射必须在缓冲一致的内存. 一致的映射建立和使用可能是昂贵的.

Streaming DMA mappings

流 DMA 映射. 流映射常常为一个单个操作建立. 一些体系当使用流映射时允许大的优化, 如我们所见, 但是这些映射也服从一个更严格的关于如何存取它们的规则. 内核开发者建议使用一致映射而不是流映射在任何可能的时候. 这个建议有 2 个原因. 第一个, 在支持映射寄存器的系统上, 每个 DMA 映射在总线上使用它们一个或多个. 一致映射, 有长的生命周期, 可以长时间独占这些寄存器, 甚至当它们不在使用时. 另外一个原因是, 在某些硬件上, 流映射可以用无法在一致映射中使用的方法来优化.

这 2 种映射类型必须以不同的方式操作; 是时候看看细节了.

15.4.4.3. 建立一致 DMA 映射

一个驱动可以建立一个一致映射，使用对 dma_alloc_coherent 的调用:

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
*dma_handle, int flag);
```

这个函数处理缓冲的分配和映射. 前 2 个参数是设备结果和需要的缓冲大小. 这个函数返回 DMA 映射的结果在 2 个地方. 来自这个函数的返回值是缓冲的一个内核虚拟地址，它可被驱动使用; 其间相关的总线地址在 dma_handle 中返回. 分配在这个函数中被处理以至缓冲被放置在一个可以使用 DMA 的位置; 常常地内存只是使用 get_free_pages 来分配(但是注意大小是以字节计的，而不是一个 order 值). flag 参数是通常的 GFP_ 值来描述内存如何被分配; 常常应当是 GFP_KERNEL (常常) 或者 GFP_ATOMIC (当在原子上下文中运行时).

当不再需要缓冲(常常在模块卸载时)，它应当被返回给系统，使用 dma_free_coherent:

```
void dma_free_coherent(struct device *dev, size_t size,
 void *vaddr, dma_addr_t dma_handle);
```

注意，这个函数象许多通常的 DMA 函数，需要提供所有的大小，CPU 地址，和 总线地址参数.

15.4.4.4. DMA 池

一个 DMA 池 是分配小的，一致 DMA 映射的分配机制. 从 dma_alloc_coherent 获得的映射可能有一页的最小大小. 如果你的驱动需要比那个更小的 DMA 区域，你应当可能使用一个 DMA 池. DMA 池也在这种情况下有用，当你可能试图对嵌在一个大结构中的小区域进行 DMA 操作. 一些非常模糊的驱动错误已被追踪到缓存一致性问题，在靠近小 DMA 区域的结构成员. 为避免这个问题，你应当一直明确分配进行 DMA 操作的区域，和其他的非 DMA 数据结构分开.

DMA 池函数定义在 <linux/dmapool.h>.

一个 DMA 池必须在使用前创建，使用一个调用:

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
 size_t size, size_t align,
 size_t allocation);
```

这里，name 是池的名子，dev 是你的设备结构，size 是要从这个池分配的缓冲区大小，align 是来自池的分配要求的硬件对齐(以字节表达的)，以及 allocation 是，如果非零，一个分配不应当越过的内存边界. 如果 allocation 以 4096 传递，例如，从池分配的缓冲不越过 4-KB 边界.

当你用完一个池，可被释放，用:

```
void dma_pool_destroy(struct dma_pool *pool);
```

你应当返回所有的分配给池, 在销毁它之前. 分配被用 dma_pool_alloc 处理:

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t *handle);
```

对这个调用, mem_flags 是常用的 GFP_ 分配标志的设置. 如果所有都进行顺利, 一个内存区(大小是当池创建时指定的)被分配和返回. 至于 dam_alloc_coherent, 结果 DMA 缓冲地址被返回作为一个内核虚拟地址, 并作为一个总线地址被存于 handle.

不需要的缓冲应当返回池, 使用:

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

15.4.4.5. 建立流 DMA 映射

流映射比一致映射有更复杂的接口, 有几个原因. 这些映射行为使用一个由驱动已经分配的缓冲, 因此, 必须处理它们没有选择的地址. 在一些体系上, 流映射也可以有多个不连续的页和多部分的"发散/汇聚"缓冲. 所有这些原因, 流映射有它们自己的一套映射函数.

当建立一个流映射时, 你必须告知内核数据移向哪个方向. 一些符号(enum dam_data_direction 类型)已为此定义:

DMA_TO_DEVICE
DMA_FROM_DEVICE

> 这 2 个符号应当是自解释的. 如果数据被发向这个设备(相应地, 也许, 到一个 write 系统调用), DMA_IO_DEVICE 应当被使用; 去向 CPU 的数据, 相反, 用 DMA_FROM_DEVICE 标志.

DMA_BIDIRECTIONAL

> 如果数据被在任一方向移动, 使用 DMA_BIDIRECTIONAL.

DMA_NONE

> 这个符号只作为一个调试辅助而提供. 试图使用带这个方向的缓冲导致内核崩溃.

可能在所有时间里试图只使用 DMA_BIDIRECTIONAL, 但是驱动作者应当抵挡住这个诱惑. 在一些体系上, 这个选择会有性能损失.

当你有单个缓冲要发送, 使用 dma_map_single 来映射它:

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);
```

返回值是总线地址, 你可以传递到设备, 或者是 NULL 如果有错误.

一旦传输完成，映射应当用 dma_unmap_single 来删除：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
enum dma_data_direction direction);
```

这里，size 和 direction 参数必须匹配那些用来映射缓冲的.

一些重要的规则适用于流 DMA 映射：

- 缓冲必须用在只匹配它被映射时给定的方向的传输.
- 一旦一个缓冲已被映射，它属于这个设备，不是处理器. 直到这个缓冲已被去映射，驱动不应当以任何方式触动它的内容. 只在调用 dma_unmap_single 后驱动才可安全存取缓冲的内容(有一个例外，我们马上见到). 其他的事情，这个规则隐含一个在被写入设备的缓冲不能被映射，直到它包含所有的要写的数据.
- 这个缓冲必须不被映射，当 DMA 仍然激活，否则肯定会有严重的系统不稳定.

你可能奇怪为什么一旦一个缓冲已被映射驱动就不能再使用它. 为什么这个规则有意义实际上有 2 个原因. 第一，当一个缓冲为 DMA 而被映射，内核必须确保缓冲中的所有的数据实际上已被写入内存. 有可能一些数据在处理器的缓存当 dma_unmap_single 被调用时，并且必须被明确刷新. 被处理器在刷新后写入缓冲的数据可能对设备不可见.

第二，考虑一下会发生什么，当被映射的缓冲在一个对设备不可存取的内存区. 一些体系在这种情况下完全失败，但是其他的创建一个反弹缓冲. 反弹缓冲只是一个分开的内存区，它对设备可存取. 如果一个缓冲被映射使用 DMA_TO_DEVICE 方向，并且要求一个反弹缓冲，原始缓冲的内容作为映射操作的一部分被拷贝. 明显地，在拷贝后的对原始缓冲的改变设备见不到. 类似地，DMA_FROM_DEVICE 反弹缓冲被 dma_unmap_single 拷回到原始缓冲；来自设备的数据直到拷贝完成才出现.

偶然地，为什么获得正确方向是重要的，反弹缓冲是一个原因. DMA_BIDIRECTIONAL 反弹缓冲在操作前后被拷贝，这常常是一个 CPU 周期的不必要浪费.

偶尔一个驱动需要存取一个流 DMA 缓冲的内容而不映射它. 已提供了一个调用来做这个：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t
size, enum dma_data_direction direction);
```

这个函数应当在处理器存取一个流 DMA 缓冲前调用. 一旦已做了这个调用，CPU "拥有" DMA 缓冲并且可以按需使用它. 在设备存取这个缓冲前，但是，拥有权应当传递回给它，使用：

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
size_t size, enum dma_data_direction direction);
```

处理器，再一次，在调用这个之后不应当存取 DMA 缓冲.

15.4.4.6. 单页流映射

偶然地，你可能想建立一个缓冲的映射，这个缓冲你有一个 struct page 指针；例如，这可能发生在使用 get_user_pages 映射用户缓冲. 为建立和取消流映射使用 struct page 指针，使用下面:

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
 unsigned long offset, size_t size,
 enum dma_data_direction direction);
void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
 size_t size, enum dma_data_direction direction);
```

offset 和 size 参数可被用来映射页的部分. 但是，建议部分页映射应当避免，除非你真正确信你在做什么. 映射一页的部分可能导致缓存一致性问题，如果这个分配只覆盖一个缓存线的一部分；这，随之，会导致内存破坏和严重的难以调试的错误.

15.4.4.7. 发散/汇聚映射

发散/汇聚映射是一个特殊类型的流 DMA 映射. 假设你有几个缓冲，都需要传送数据到或者从设备. 这个情况可来自几个方式，包括从一个 readv 或者 writev 系统调用，一个成簇的磁盘 I/O 请求，或者一个页链表在一个被映射的内核 I/O 缓冲. 你可简单地映射每个缓冲，轮流的，并且进行要求的操作，但是有几个优点来一次映射整个链表.

许多设备可以接收一个散布表数组指针和长度，并且传送它们全部在一个 DMA 操作中；例如，"零拷贝"网络是更轻松如果报文在多个片中建立. 另一个映射发散列表为一个整体的理由是利用在总线硬件上有映射寄存器的系统. 在这样的系统上，物理上不连续的页从设备的观点看可被汇集为一个单个的，连续的数组. 这个技术只当散布表中的项在长度上等于页大小(除了第一个和最后一个)，但是当它做这个工作时，它可转换多个操作到一个单个的 DMA，和有针对性的加速事情.

最后，如果一个反弹缓冲必须被使用，应该连接整个列表为一个单个缓冲(因为它在被以任何方式拷贝).

因此现在你确信散布表的映射在某些情况下是值得的. 映射一个散布表的第一步是创建和填充一个 struct scatterlist 数组，它描述被传输的缓冲. 这个结构是体系依赖的，并且在 <asm/scatterlist.h> 中描述. 但是，它常常包含 3 个成员:

struct page *page;

  struct page 指针，对应在发散/汇聚操作中使用的缓冲.

unsigned int length;
unsigned int offset;

  缓冲的长度和它的页内偏移.

为映射一个发散/汇聚 DMA 操作, 你的驱动应当设置 page, offset, 和 length 成员在一个 struct scatterlist 项给每个要被发送的缓冲. 接着调用:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction)
```

这里 nents 是传入的散布表项的数目. 返回值是要发送的 DMA 缓冲的数目. 它可能小于 nents.

对于输入散布表中的每个缓冲, dma_map_sg 决定了正确的给设备的总线地址. 作为任务的一部分, 它也连接在内存中相近的缓冲. 如果你的驱动运行的系统有一个 I/O 内存管理单元, dma_map_sg 也编程这个单元的映射寄存器, 可能的结果是, 从你的驱动的观点, 你能够传输一个单个的, 连续的缓冲. 你将不会知道传送的结果将看来如何, 但是, 直到在调用之后.

你的驱动应当传送由 pci_map_sg 返回的每个缓冲. 总线地址和每个缓冲的长度存储于 struct scatterlist 项, 但是它们在结构中的位置每个体系不同. 2 个宏定义已被定义来使得可能编写可移植的代码:

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
```

从这个散布表入口返回总线(DMA)地址.

```
unsigned int sg_dma_len(struct scatterlist *sg);
```

返回这个缓冲的长度.

再次, 记住要传送的缓冲的地址和长度可能和传递给 dma_map_sg 的不同.

一旦传送完成, 一个 发散/汇聚 映射被使用 dma_unmap_sg 去映射:

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents,
enum dma_data_direction direction);
```

注意 nents 必须是你起初传递给 dma_map_sg 的入口项的数目, 并且不是这个函数返回给你的 DMA 缓冲的数目.

发散/汇聚映射是流 DMA 映射, 并且同样的存取规则如同单一映射一样适用. 如果你必须存取一个被映射的发散/汇聚列表, 你必须首先同步它:

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
 int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
 int nents, enum dma_data_direction direction);
```

15.4.4.8. PCI 双地址周期映射

正常地，DMA 支持层使用 32-位 总线地址，可能受限于一个特定设备的 DMA 掩码. PCI 总线，但是，也支持一个 64-位地址模式，双地址周期(DAC). 通常的 DMA 层不支持这个模式，因为几个理由，第一个是它是一个 PCI-特定 的特性. 还有，许多 DAC 的实现满是错误，并且，因为 DAC 慢于一个常规的，32-位 DMA，可能有一个性能开销. 即便如此，有的应用程序使用 DAC 是正确的事情；如果你有一个设备可能使用非常大的位于高内存的缓冲，你可能要考虑实现 DAC 支持. 这个支持只对 PCI 总线适用，因此 PCI-特定的函数必须被使用.

为使用 DAC，你的驱动必须包含 <linux/pci.h>. 你必须设置一个单独的 DMA 掩码:

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

你可使用 DAC 寻址只在这个调用返回 0 时. 一个特殊的类型 (dma64_addr_t) 被用作 DAC 映射. 为建立一个这些映射，调用 pci_dac_page_to_dma:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
unsigned long offset, int direction);
```

DAC 映射，你将注意到，可能被完成只从 struct page 指针(它们应当位于高内存，毕竟，否则使用它们没有意义了)；它们必须一次一页地被创建. direction 参数是在通用 DMA 层中使用的 enum dma_data_direction 的 PCI 对等体；它应当是 PCI_DMA_TODEVICE，PCI_DMA_FROMDEVICE，或者 PCI_DMA_BIRDIRECTIONAL.

DAC 映射不要求外部资源，因此在使用后没有必要明确释放它们. 但是，有必要象对待其他流映射一样对待 DAC 映射，并且遵守关于缓冲所有权的规则. 有一套函数来同步 DMA 缓冲，和通常的变体相似:

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
 dma64_addr_t dma_addr,
 size_t len,
 int direction);

void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
 dma64_addr_t dma_addr,
 size_t len,
 int direction);
```

15.4.4.9. 一个简单的 PCI DMA 例子

作为一个 DMA 映射如何被使用的例子，我们展示了一个简单的给一个 PCI 设备的 DMA 编码的例子. 在 PCI 总线上的数据的 DMA 操作的形式非常依赖被驱动的设备. 因此，这个例子不适用于任何真实的设备；相反，它是一个称为 dad (DMA Acquisiton Device) 的假想驱动的一部分. 一个给这个设备的驱动可能定义一个传送函数象这样:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
```

```
 size_t count)
{
 dma_addr_t bus_addr;

 /* Map the buffer for DMA */
 dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
 dev->dma_size = count;
 bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,

 dev->dma_dir);
 dev->dma_addr = bus_addr;

 /* Set up the device */
 writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
 writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
 writel(dev->registers.addr, cpu_to_le32(bus_addr));
 writel(dev->registers.len, cpu_to_le32(count));

 /* Start the operation */
 writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
 return 0;

}
```

这个函数映射要被传送的缓冲并且启动设备操作. 这个工作的另一半必须在中断服务过程中完成, 这个看来如此:

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
 struct dad_dev *dev = (struct dad_dev *) dev_id;

 /* Make sure it's really our device interrupting */
 /* Unmap the DMA buffer */
 dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
 dev->dma_size, dev->dma_dir);

 /* Only now is it safe to access the buffer, copy to user, etc. */
 ...
}
```

显然, 这个例子缺乏大量的细节, 包括可能需要的任何步骤来阻止启动多个同时的 DMA 操作.

15.4.5. ISA 设备的 DMA

ISA 总线允许 2 类 DMA 传送: 本地 DMA 和 ISA 总线主 DMA. 本地 DMA 使用在主板上的标准 DMA-控制器电路来驱动 ISA 总线上的信号线. ISA 总线主 DMA, 另一方面, 完全由外设处理, 至少从驱动的观点看. 一个 ISA 总线主的例子是 1542 SCSI 控制器, 在内核源码中是在 drivers/scsi/aha1542.c.

至于本地 DMA, 有 3 个实体包含在 ISA 总线上的 DMA 数据传送.

The 8237 DMA controller (DMAC)

控制器持有关于 DMA 传送的信息, 诸如方向, 内存地址, 以及传送的大小. 它还包含一个计数器来跟踪进行中的传送的状态. 当这个控制器收到一个 DMA 请求信号, 它获得总线的控制权并且驱动信号线以便设备可读或些它的数据.

The peripheral device

这个设备必须激活 DMA 请求线当它准备传送数据时. 实际的传送由 DMAC 管理; 硬件设备顺序读或写数据到总线当控制器探测设备时. 设备常常触发中断当传送结束时.

The device driver

这个驱动什么不做; 它提供给 DMA 控制器方向, 总线地址,和传送的大小. 它还和它的外设通讯来准备传送数据和响应中断当 DMA 结束时.

开始的在 PC 上使用的 DMA 控制器管理 4 个"通道", 每个有一套 DMA 寄存器. 4 个设备可同时存储它们的 DMA 信息在控制器中. 更新的 PC 包含相同的 2 个 DMAC 设备[51]: 第 2 个控制器(主)被连接到系统的处理器, 并且第 1 个(从)被连接到第 2 个控制器的通道 0.

最初的 PC 只有一个控制器; 第 2 个是在基于 286 的平台上增加的. 但是, 第 2 个控制器如同主控制器一样被连接, 因为它处理 16-位的传送; 第 1 个只传送 8 位每次并且它为向后兼容而存在.

通道的编号从 0 到 7: 通道 4 对 ISA 外设不可用, 因为它在内部用来层叠从控制器到主控制器. 因此, 可用的通道是 0 到 3 在从控制器上(8-位 通道) 和 5 到 7 到主控制器上(16-位通道). 任何 DMA 传送的大小, 当被存储于控制器中, 是一个代表总线周期的数目的 16-位数. 最大的传送大小是, 因此, 64KB 对于从控制器(因为它传送 8 位在一个周期)和 128KB 对于主控制器(它进行 16-位 传送).

因为 DMA 控制器是一个系统范围的资源, 内核帮助处理这个. 它使用一个 DMA 注册来提供一个请求并释放机制给 DMA 通道, 和一套函数来在 DMA 控制器中配置通道信息.

15.4.5.1. 注册 DMA 使用

你应当熟悉内核注册 -- 我们已经见到它们在 I/O 端口和中断线. DMA 通道注册和其他的类似. 在 <asm/dma.h> 中已经包含, 下面的函数可用来获得和释放一个 DMA 通道的拥有权:

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

通道参数是一个在 0 到 7 之间的数, 更精确些, 一个小于 MAX_DMA_CHANNELS 的正值. 在 PC 上, MAX_DMA_CHANNELS 定义为 8 来匹配硬件. name 参数是一个字符串来标识设备. 特定的 name 出现在文件 /proc/dma, 它可被用户程序读.

从 request_dma 的返回值是 0 对于成功, 是 -EINVAL 或者 -EBUSY 如果有错误. 前者意思是请求的通道超范围, 后者意思是另一个设备持有这个通道.

我们推荐你象对待 I/O 端口和中断线一样小心对待 DMA 通道; 在打开时请求通道好于从模块初始化函数里请求它. 延后请求允许在驱动之间的一些共享; 例如, 你的声卡和模拟 I/O 接口可以共享 DMA 通道只要它们不同时使用.

我们还建议你请求 DMA 通道在你已请求中断线之后并且你在中断前释放它. 这是惯用的顺序来请求这 2 个资源; 遵循这个惯例避免了死锁的可能. 注意每个使用 DMA 的设备需要一个 IRQ 线; 否则, 它不能指示数据传送的完成.

在一个典型的情况, open 代码看来如下, 引用了我们的假想的 dad 模块. dad 设备使用了一个快速中断处理, 不带共享 IRQ 线支持.

```
int dad_open (struct inode *inode, struct file *filp)
{

 struct dad_device *my_device;

/* ... */
 if ( (error = request_irq(my_device.irq, dad_interrupt,
 SA_INTERRUPT, "dad", NULL)) )
 return error; /* or implement blocking open */

 if ( (error = request_dma(my_device.dma, "dad")) ) {
 free_irq(my_device.irq, NULL);
 return error; /* or implement blocking open */
 }

 /* ... */
 return 0;
}
```

和 open 匹配的 close 实现看来如此:

```
void dad_close (struct inode *inode, struct file *filp)
{

 struct dad_device *my_device;
 /* ... */
 free_dma(my_device.dma);
 free_irq(my_device.irq, NULL);
 /* ... */
}
```

这是 /proc/dma 文件 在一个安装有声卡的系统中的样子:

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

注意, 缺省的声音驱动获得 DMA 通道在系统启动时并且从不释放它. 层叠的入口是一个占位者, 指出通道 4 对驱动不可用, 如同前面解释的.

15.4.5.2. 和 DMA 控制器通讯

在注册后, 驱动工作的主要部分包括配置 DMA 控制器正确操作. 这个任务并非微不足道的, 但是幸运的是, 内核输出了典型驱动需要的所有的函数.

驱动需要配置 DMA 控制器或者读或写被调用时, 或者当准备异步传送时. 后面这个任务或者在打开时进行或者响应一个 ioctl 命令, 根据驱动和它实现的策略. 这里展示的代码是典型地被读或写设备方法调用的.

这一小节提供一个对于 DMA 控制器内部的快速概览, 这样你可理解这里介绍的代码. 如果你想知道更多, 我们劝你读 <asm/dma.h> 和一些描述 PC 体系的硬件手册. 特别地, 我们不处理 8-位 和 16-位 传送的问题. 如果你在编写设备驱动给 ISA 设备板, 你应当在设备的硬件手册中找到相关的信息.

DMA 控制器是一个共享的资源, 并且如果多个处理器试图同时对它编程会引起混乱. 为此, 控制器被一个自旋锁保护, 称为 dma_spin_lock. 驱动不应当直接操作这个锁; 但是, 2 个函数已提供给你来做这个:

```
unsigned long claim_dma_lock( );
```

获取 DMA 自旋锁. 这个函数还在本地处理器上阻塞中断; 因此, 返回值是一些描述之前中断状态的标志; 它必须被传递给随后的函数来恢复中断状态, 当你用完这个锁.

```
void release_dma_lock(unsigned long flags);
```

返回 DMA 自旋锁并且恢复前面的中断状态.

自旋锁应当被持有, 当使用下面描述的函数时. 但是, 它不应当被持有, 在实际的 I/O 当中. 一个驱动应当从不睡眠当持有一个自旋锁时.

必须被加载到控制器中的信息包括 3 项: RAM 地址, 必须被传送的原子项的数目(以字节或字计), 以及传送的方向. 为此, 下列函数由 <asm/dma.h> 输出:

```
void set_dma_mode(unsigned int channel, char mode);
```

指示是否这个通道必须从设备读(DMA_MODE_READ)或者写到设备(DMA_MODE_WRITE). 存在第 3 个模式, DMA_MODE_CASCADE, 它被用来释放对总线的控制. 层叠是第 1 个控制器连接到第 2 个控制器顶部的方式, 但是它也可以被真正的 ISA 总线主设备使用. 我们这里不讨论总线控制.

```
void set_dma_addr(unsigned int channel, unsigned int addr);
```

分配 DMA 缓冲的地址. 这个函数存储 addr 的低 24 有效位在控制器中. addr 参数必须是一个总线地址(见"总线地址"一节, 在本章前面).

void set_dma_count(unsigned int channel, unsigned int count);

分配传送的字节数. count 参数也表示给 16-位 通道的字节; 在这个情况下, 这个数必须是偶数.

除了这些函数, 有一些维护工具必须用, 当处理 DMA 设备时:

void disable_dma(unsigned int channel);

一个 DMA 通道可在控制器内部被关闭. 这个通道应当在控制器被配置为阻止进一步不正确的操作前被关闭. (否则, 会因为控制器被通过 8-位数据传送被编程而发生破坏, 并且, 因此, 之前的功能都不自动执行.

void enable_dma(unsigned int channel);

这个函数告知控制器 DMA 通道包含有效数据.

int get_dma_residue(unsigned int channel);

这个驱动有时需要知道是否一个 DMA 传输已经完成. 这个函数返回仍要被传送的字节数. 在一次成功的传送后的返回值是 0 并且在控制器在工作时是不可预测的(但不是 0). 这种不可预测性来自需要通过 2 个 8-位输入操作来获得 16-位 的余数.

void clear_dma_ff(unsigned int channel) ;

这个函数清理 DMA flip-flop. 这个 flip-flop 用来控制对 16-位 寄存器的存取. 这些寄存器被 2 个连续的 8-位操作来存取, 并且这个 flip-flop 被用来选择低有效字节(当它被清零)或者是最高有效字节(当它被置位). flip-flop 自动翻转当已经传送了 8 位; 程序员必须清除 flip-flop(来设置它为已知的状态)在存取 DMA 寄存器之前.

使用这些, 一个驱动可如下实现一个函数来准备一次 DMA 传送:

```
int dad_dma_prepare(int channel, int mode, unsigned int buf, unsigned int count)
{
 unsigned long flags;

    flags = claim_dma_lock();
 disable_dma(channel);
 clear_dma_ff(channel);
 set_dma_mode(channel, mode);
 set_dma_addr(channel, virt_to_bus(buf));
 set_dma_count(channel, count);
 enable_dma(channel);
 release_dma_lock(flags);
```

```
 return 0;
}
```

接着，一个象下一个的函数被用来检查 DMA 的成功完成:

```
int dad_dma_isdone(int channel)
{

int residue;
    unsigned long flags = claim_dma_lock ();
 residue = get_dma_residue(channel);
 release_dma_lock(flags);
    return (residue == 0);
}
```

未完成的唯一一个事情是配置设备板. 这个设备特定的任务常常包含读或写几个 I/O 端口. 设备在几个大的方面不同. 例如, 一些设备期望程序员告诉硬件 DMA 缓冲有多大, 并且有时驱动不得不读一个被硬连到设备中的值. 为配置板, 硬件手册是你唯一的朋友.

---

[49] 当然, 什么事情都有例外; 见"接收中断缓解"一节在 17 章, 演示了高性能网络驱动如何被使用轮询最好地实现.

[50] 碎片一词常常用于磁盘来表达文件没有连续存储在磁介质上. 相同的概念适用于内存, 这里每个虚拟地址空间在整个物理 RAM 散布, 并且难于获取连续的空闲页当请求一个 DMA 缓冲.

[51] 这些电路现在是主板芯片组的一部分, 但是几年前它们是 2 个单独的 8237 芯片.

15.5. 快速参考

本章介绍了下列关于内存处理的符号:

15.5.1. 介绍性材料

```
#include <linux/mm.h>
#include <asm/page.h>
```

和内存管理相关的大部分函数和结构, 原型和定义在这些头文件.

```
void *__va(unsigned long physaddr);
unsigned long __pa(void *kaddr);
```

在内核逻辑地址和物理地址之间转换的宏定义.

```
PAGE_SIZE
PAGE_SHIFT
```

常量，给出底层硬件的页的大小(字节)和一个页面号必须被移位来转变为一个物理地址的位数.

```
struct page
```

在系统内存映射中表示一个硬件页的结构.

```
struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
struct page *pfn_to_page(int pfn);
```

宏定义，在内核逻辑地址和它们相关的内存映射入口之间转换的. page_address 只用在低地址页或者已被明确映射的高地址页. pfn_to_page 转换一个页面号到它的相关的 struct page 指针.

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 返回一个内核虚拟地址，被映射到给定页，如果需要并创建映射. kunmap 为给定页删除映射.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap 的高性能版本；结果的映射只能被原子代码持有. 对于驱动，type 应当是 KM_USER1，KM_USER1，KM_IRQ0，或者 KM_IRQ1.

```
struct vm_area_struct;
```

描述一个 VMA 的结构.

15.5.2. 实现 mmap

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add,
unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add,
unsigned long phys_add, unsigned long size, pgprot_t prot);
```

位于 mmap 核心的函数. 它们映射 size 字节的物理地址，从 pfn 指出的页号开始到虚拟地址 virt_add. 和虚拟空间相关联的保护位在 prot 里指定. io_remap_page_range 应当在目标地址在 I/O 内存空间里时被使用.

```
struct page *vmalloc_to_page(void *vmaddr);
```

转换一个由 vmalloc 获得的内核虚拟地址到它的对应的 struct page 指针.

15.5.3. 实现直接 I/O

```
int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start, int len, int write, int force, struct page **pages, struct vm_area_struct **vmas);
```

函数，加锁一个用户空间缓冲到内存并且返回对应的 struct page 指针. 调用者必须持有 mm->mmap_sem.

```
SetPageDirty(struct page *page);
```

宏定义，标识给定的页为"脏"(被修改)并且需要写到它的后备存储，在它被释放前.

```
void page_cache_release(struct page *page);
```

释放给定的页从页缓存中.

```
int is_sync_kiocb(struct kiocb *iocb);
```

宏定义，返回非零如果给定的 IOCB 需要同步执行.

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

函数，指示一个异步 I/O 操作完成.

15.5.4. 直接内存存取

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

过时的不好的函数，在内核，虚拟，和总线地址之间转换. 总线地址必须用来和外设通讯.

```
#include <linux/dma-mapping.h>
```

需要来定义通用 DMA 函数的头文件.

```
int dma_set_mask(struct device *dev, u64 mask);
```

对于无法寻址整个 32-位范围的外设，这个函数通知内核可寻址的地址范围并且如果可进行 DMA 返回非零.

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *bus_addr, int flag);
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr, dma_handle_t bus_addr);
```

分配和释放一致 DMA 映射，对一个将持续在驱动的生命周期中的缓冲.

```
#include <linux/dmapool.h>
```

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev, size_t size, size_t
align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```

创建, 销毁, 和使用 DMA 池来管理小 DMA 区的函数.

```
enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE
```

符号, 用来告知流映射函数在什么方向数据移入或出缓冲.

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum
dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size, enum
dma_data_direction direction);
```

创建和销毁一个单使用, 流 DMA 映射.

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t size, enum
dma_data_direction direction);
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr, size_t size,
enum dma_data_direction direction);
```

同步一个由一个流映射的缓冲. 必须使用这些函数, 如果处理器必须存取一个缓冲当使用流映射时.(即, 当设备拥有缓冲时).

```
#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

这个散布表结构描述一个涉及不止一个缓冲的 I/O 操作. 宏 sg_dma_address he sg_dma_len 可用来抽取总线地址和缓冲长度来传递给设备, 当实现发散/汇聚操作时.

```
dma_map_sg(struct device *dev, struct scatterlist *list, int nents, enum
dma_data_direction direction);
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum
dma_data_direction direction);
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction);
```

dma_map_sg 映射一个 发散/汇聚 操作, 并且 dma_unmap_sg 恢复这些映射. 如果在这个映射被激活时缓冲必须被存取, dma_sync_sg_* 可用来同步.

/proc/dma

包含在 DMA 控制器中的被分配的通道的文本快照的文件. 基于 PCI 的 DMA 不显示, 因为每个板独立工作, 不需要分配一个通道在 DMA 控制器中.

#include <asm/dma.h>

定义或者原型化所有和 DMA 相关的函数和宏定义. 它必须被包含来使用任何下面符号.

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

存取 DMA 注册. 注册必须在使用 ISA DMA 通道之前进行.

```
unsigned long claim_dma_lock( );
void release_dma_lock(unsigned long flags);
```

获取和释放 DMA 自旋锁, 它必须被持有, 在调用其他的在这个列表中描述的 ISA DMA 函数之前. 它们在本地处理器上也关闭和重新使能中断

```
void set_dma_mode(unsigned int channel, char mode);
void set_dma_addr(unsigned int channel, unsigned int addr);
void set_dma_count(unsigned int channel, unsigned int count);
```

编程 DMA 信息在 DMA 控制器中. addr 是一个总线地址.

```
void disable_dma(unsigned int channel);
void enable_dma(unsigned int channel);
```

一个 DMA 通道必须被关闭在配置期间. 这些函数改变 DMA 通道的状态.

```
int get_dma_residue(unsigned int channel);
```

如果这驱动需要知道一个 DMA 传送在进行, 它可调用这个函数, 返回尚未完成的数据传输的数目. 在成功的 DMA 完成后, 这个函数返回 0; 值是不可预测的当数据仍然在传送时.

```
void clear_dma_ff(unsigned int channel);
```

DMA flip-flop 被控制器用来传送 16-位值, 通过 2 个 8 位操作. 它必须被清除, 在发送任何数据给处理器之前.

第 16 章 块驱动

至今，我们的讨论一直限于字符驱动. 但是，在 Linux 系统中有其他类型的驱动，并且到时候要开阔我们的视野了. 因此，本章讨论块驱动.

一个块驱动提供设备的存取，这个设备可随机地以固定大小的块传送数据--主要的是，磁盘驱动. Linux 内核看待块设备根本上不同于字符设备；结果，块驱动有明显不同的接口和它们自己的特殊的挑战.

高效的块驱动对于性能是重要的 -- 不只是为在用户应用程序的明确的读和写. 现代的有虚拟内存的系统将不需要的数据移向(希望地)二级存储中，它常常是一个磁盘驱动器. 块驱动是核心内存和二级存储之间的导管；因此，它们可组成虚拟内存子系统的一部分. 虽然可能编写一个块驱动不必知道 struct page 和其他重要的内存概念，任何需要编写一个高性能驱动的人必须使用 15 章所涉及的内容.

许多块层的设计围绕性能. 许多字符设备可在它们的最大速率以下运行，并且系统的总体性能不被影响. 但是如果它的块 I/O 子系统没有调整好，系统不能很好地运行. Linux 块驱动接口允许你从一个块设备中获得最多输出，但是有必要，施加一些你必须处理的复杂性. 好的是，2.6 的块接口比之前的内核很大提高.

如你会期望的，本章的讨论集中在一个例子驱动，它实现了一个面向块的，基于内存的设备. 基本上，它是一个 ramdisk. 内核硬件包含了一个很高级的 ramdisk 实现，但是我们的驱动(称为 sbull)让我们演示创建一个块驱动，同时最小化无关的复杂性.

在进入细节之前，我们精确定义几个词语. 一个块是一个固定大小的数据块，大小由内核决定. 块常常是 4096 字节，但是这个值可依赖体系和使用的文件系统而变化. 一个扇区，相反，是一个小块，它的大小常常由底层的硬件决定. 内核期望处理实现 512-字节扇区的设备. 如果你的设备使用不同的大小，内核调整并且避免产生硬件无法处理的 I/O 请求. 但是，它值得记住，任何时候内核给你一个扇区号，它是工作在一个 512-字节扇区的世界. 如果你使用不同的硬件扇区大小，你必须相应地调整内核的扇区号. 我们在 sbull 驱动中见如何完成这个.

16.1. 注册

块驱动，象字符驱动，必须使用一套注册接口来使内核可使用它们的设备. 概念是类似的，但是块设备注册的细节是都不同的. 你有一整套新的数据结构和设备操作要学习.

16.1.1. 块驱动注册

大部分块驱动采取的第一步是注册它们自己到内核. 这个任务的函数是 register_blkdev(在 <linux/fs.h> 中定义):

```
int register_blkdev(unsigned int major, const char *name);
```

参数是你的设备要使用的主编号和关联的名子(内核将显示它在 /proc/devices). 如果 major 传递为 0，内核分配一个新的主编号并且返回它给调用者. 如常，自 register_blkdev 的一个负的返回值指示已发生了一个错误.

取消注册的对应函数是:

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，参数必须匹配传递给 register_blkdev 的那些，否则这个函数返回 -EINVAL 并且什么都不注销.

在 2.6 内核，对 register_blkdev 的调用完全是可选的. 由 register_blkdev 所进行的功能已随时间正在减少；这个调用唯一的任务是 (1) 如果需要，分配一个动态主编号，并且 (2) 在 /proc/devices 创建一个入口. 在将来的内核，register_blkdev 可能被一起去掉. 同时，但是，大部分驱动仍然调用它；它是惯例.

16.1.2. 磁盘注册

虽然 register_blkdev 可用来获得一个主编号，它不使任何磁盘驱动器对系统可用. 有一个分开的注册接口你必须使用来管理单独的驱动器. 使用这个接口要求熟悉一对新结构，这就是我们的起点.

16.1.2.1. 块设备操作

字符设备通过 file_ 操作结构使它们的操作对系统可用. 一个类似的结构用在块设备上；它是 struct block_device_operations，定义在 <linux/fs.h>. 下面是一个对这个结构中的成员的简短的概览；当我们进入 sbull 驱动的细节时详细重新访问它们.

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

就像它们的字符驱动对等体一样工作的函数；无论何时设备被打开和关闭都调用它们. 一个字符驱动可能通过启动设备或者锁住门(为可移出的介质)来响应一个 open 调用. 如果你将介质锁入设备，你当然应当在 release 方法中解锁.

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);
```

实现 ioctl 系统调用的方法. 但是，块层首先解释大量的标准请求；因此大部分的块驱动 ioctl 方法相当短.

```
int (*media_changed) (struct gendisk *gd);
```

被内核调用来检查是否用户已经改变了驱动器中的介质的方法，如果是这样返回一个非零值. 显然，这个方法仅适用于支持可移出的介质的驱动器(并且最好给驱动一个"介质被改变"标志)；在其他情况下可被忽略.

struct gendisk 参数是内核任何表示单个磁盘；我们将在下一节查看这个结构.

```
int (*revalidate_disk) (struct gendisk *gd);
```

revalidate_disk 方法被调用来响应一个介质改变；它给驱动一个机会来进行需要的任何工作使新介质准备好使用. 这个函数返回一个 int 值，但是值被内核忽略.

struct module *owner;

一个指向拥有这个结构的模块的指针; 它应当常常被初始化为 THIS_MODULE.

专心的读者可能已注意到这个列表一个有趣的省略: 没有实际读或写数据的函数. 在块 I/O 子系统, 这些操作由请求函数处理, 它们应当有它们自己的一节并且在本章后面讨论. 在我们谈论服务请求之前, 我们必须完成对磁盘注册的讨论.

16.1.2.2. gendisk 结构

struct gendisk (定义于 <linux/genhd.h>) 是单独一个磁盘驱动器的内核表示. 事实上, 内核还使用 gendisk 来表示分区, 但是驱动作者不必知道这点. struct gedisk 中有几个成员, 必须被一个块驱动初始化:

int major;
int first_minor;
int minors;

描述被磁盘使用的设备号的成员. 至少, 一个驱动器必须使用最少一个次编号. 如果你的驱动会是可分区的, 但是(并且大部分应当是), 你要分配一个次编号给每个可能的分区. 次编号的一个普通的值是 16, 它允许"全磁盘"设备盒 15 个分区. 一些磁盘驱动使用 64 个次编号给每个设备.

char disk_name[32];

应当被设置为磁盘驱动器名子的成员. 它出现在 /proc/partitions 和 sysfs.

struct block_device_operations *fops;

来自前一节的设备操作集合.

struct request_queue *queue;

被内核用来管理这个设备的 I/O 请求的结构; 我们在"请求处理"一节中检查它.

int flags;

一套标志(很少使用), 描述驱动器的状态. 如果你的设备有可移出的介质, 你应当设置 GENHD_FL_REMOVABLE. CD-ROM 驱动器可设置 GENHD_FL_CD. 如果, 由于某些原因, 你不需要分区信息出现在 /proc/partitions, 设置 GENHD_FL_SUPPRESS_PARTITIONS_INFO.

sector_t capacity;

这个驱动器的容量, 以 512-字节扇区来计. sector_t 类型可以是 64 位宽. 驱动不应当直接设置这个成员; 相反, 传递扇区数目给 set_capacity.

```
void *private_data;
```

块驱动可使用这个成员作为一个指向它们自己内部数据的指针.

内核提供了一小部分函数来使用 gendisk 结构. 我们在这里介绍它们, 接着看 sbull 如何使用它们来使系统可使用它的磁盘驱动器.

struct gendisk 是一个动态分配的结构, 它需要特别的内核操作来初始化; 驱动不能自己分配这个结构. 相反, 你必须调用:

```
struct gendisk *alloc_disk(int minors);
```

minors 参数应当是这个磁盘使用的次编号数目; 注意你不能在之后改变 minors 成员并且期望事情可以正确工作. 当不再需要一个磁盘时, 它应当被释放, 使用:

```
void del_gendisk(struct gendisk *gd);
```

一个 gendisk 是一个被引用计数的结构(它含有一个 kobject). 有 get_disk 和 put_disk 函数用来操作引用计数, 但是驱动应当从不需要做这个. 正常地, 对 del_gendisk 的调用去掉了最一个 gendisk 的最终的引用, 但是不保证这样. 因此, 这个结构可能继续存在(并且你的方法可能被调用)在调用 del_gendisk 之后. 但是, 如果你删除这个结构当没有用户时(即, 在最后的释放之后, 或者在你的模块清理函数), 你可确信你不会再收到它的信息.

分配一个 gendisk 结构不能使系统可使用这个磁盘. 要做到这点, 你必须初始化这个结构并且调用 add_disk:

```
void add_disk(struct gendisk *gd);
```

这里记住一件重要的事情:一旦你调用 add_disk, 这个磁盘是"活的"并且它的方法可被在任何时间被调用. 实际上, 这样的第一个调用将可能发生, 即便在 add_disk 返回之前; 内核将读前几个字节以试图找到一个分区表. 因此你不应当调用 add_disk 直到你的驱动被完全初始化并且准备好响应对那个磁盘的请求.

16.1.3. 在 sbull 中的初始化

是时间进入一些例子了. sbull 驱动(从 O'Reilly 的 FTP 网站, 以及其他例子源码)实现一套内存中的虚拟磁盘驱动器. 对每个驱动器, sbull 分配(使用 vmalloc, 为了简单)一个内存数组; 它接着使这个数组可通过块操作来使用. 这个 sbull 驱动可通过分区这个驱动器, 在上面建立文件系统, 以及加载到系统层级中来测试.

象我们其他的例子驱动一样, sbull 允许一个主编号在编译或者模块加载时被指定. 如果没有指定, 动态分配一个. 因为对 register_blkdev 的调用被用来动态分配, sbull 应当这样做:

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0)
{
```

```
        printk(KERN_WARNING "sbull: unable to get major number\n");
        return -EBUSY;
}
```

同样，象我们在本书已展现的其他虚拟设备，sbull 设备由一个内部结构描述:

```
struct sbull_dev {
 int size;  /* Device size in sectors */
 u8 *data;  /* The data array */
 short users;  /* How many users */
 short media_change;  /* Flag a media change? */
 spinlock_t lock;  /* For mutual exclusion */
 struct request_queue *queue;  /* The device request queue */
 struct gendisk *gd;  /* The gendisk structure */
 struct timer_list timer;  /* For simulated media changes */
};
```

需要几个步骤来初始化这个结构，并且使系统可用关联的设备. 我们从基本的初始化开始，并且分配底层的内存:

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL)
{
        printk (KERN_NOTICE "vmalloc failure.\n");
        return;
}
spin_lock_init(&dev->lock);
```

重要的是在下一步之前分配和初始化一个自旋锁，下一步是分配请求队列. 我们在进入请求处理时详细看这个过程；现在，只需说必要的调用是:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

这里，sbull_request 是我们的请求函数 -- 实际进行块读和写请求的函数. 当我们分配一个请求队列时，我们必须提供一个自旋锁来控制对那个队列的存取. 这个锁由驱动提供而不是内核通常的部分，因为，常常，请求队列和其他的驱动数据结构在相同的临界区；它们可能被同时存取. 如同任何分配内存的函数，blk_init_queue 可能失败，因此你必须在继续之前检查返回值.

一旦我们有我们的设备内存和请求队列，我们可分配，初始化，并且安装对应的 gendisk 结构. 做这个工作的代码是:

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd)
{
        printk (KERN_NOTICE "alloc_disk failure\n");
        goto out_vfree;
}
dev->gd->major = sbull_major;
```

```
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

这里, SBULL_MINORS 是每个 sbull 设备所支持的次编号的数目. 当我们设置第一个次编号给每个设备, 我们必须考虑被之前的设备所用的全部编号. 磁盘的名子被设置, 这样第一个是 sbulla, 第二个是 sbullb, 等等. 用户空间可接着添加分区号以便它们在第 2 个设备上的分区可能是 /dev/sbull3.

一旦所有的都被设置, 我们以对 add_disk 的调用来结束. 我们的几个方法将在 add_disk 返回时被调用, 因此我们负责做这个调用, 这是初始化我们的设备的最后一步.

16.1.4. 注意扇区大小

如同我们之前提到的, 内核对待每个磁盘如同一个 512-字节扇区的数组. 不是所有的硬件都使用那个扇区大小, 但是. 使一个有不同扇区大小的设备工作不是一件很难的事; 只要小心处理几个细节. sbull 设备输出一个 hardsect_size 参数, 可被用来改变设备的"硬件"扇区大小. 通过看它的实现, 你可见到如何添加这个支持到你自己的驱动.

这些细节中的第一个是通知内核你的设备支持的扇区大小. 硬件扇区大小是一个在请求队列的参数, 而不是在 gendisk 结构. 这个大小通过调用 blk_queue_hardsect_size 设置的, 在分配队列后马上进行:

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

一旦完成那个, 内核坚持你的设备的硬件扇区大小. 所有的 I/O 请求被正确对齐到一个硬件扇区的起始, 并且每个请求的长度是一个整数的扇区数. 你必须记住, 但是, 内核一直以 512-字节扇区表述自己; 因此, 有必要相应地转换所有的扇区号. 因此, 例如, 当 sbull 在它的 gendisk 结构中设置设备的容量时, 这个调用看来象:

```
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
```

KERNEL_SECTOR_SIZE 是一个本地定义的常量, 我们用来调整内核的 512-字节和任何我们已被告知要使用的大小. 在我们查看 sbull 请求处理逻辑中会不时看到这类计算出来.

16.2. 块设备操作

在前面一节中我们对 block_device_operations 有了简短的介绍. 现在我们详细些看看这些操作, 在进入请求处理之前. 为此, 是时间提到 sbull 驱动的另一个特性: 它假装是一个可移出的设备. 无论何时最后一个用户关闭设备, 一个 30 秒的定时器被设置; 如果设备在这个时间内不被打开, 设备的内容被清除, 并且内核被告知介质已被改变. 30 秒延迟给了用户时间, 例如, 来卸载一个 sbull 设备在创建一个文件系统之后.

16.2.1. open 和 release 方法

为实现模拟的介质移出, 当最后一个用户已关闭设备时 sbull 必须知道. 一个用户计数被驱动维护. 它是 open 和 close 方法的工作来保持这个计数最新.

open 方法看起来非常类似于它的字符驱动对等体; 它用相关的节点和文件结构指针作为参数. 当一个节点引用一个块设备, i_bdev->bd_disk 包含一个指向关联 gendisk 结构的指针; 这个指针可用来获得一个驱动的给设备的内部数据结构. 即, 实际上, sbull open 方法做的第一件事:

```
static int sbull_open(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
        del_timer_sync(&dev->timer);
        filp->private_data = dev;
        spin_lock(&dev->lock)
        ;
        if (! dev->users)
                check_disk_change(inode->i_bdev);
        dev->users++;
        spin_unlock(&dev->lock)
        ;
        return 0;
}
```

一旦 sbull_open 有它的设备结构指针, 它调用 del_timer_sync 来去掉"介质移出"定时器, 如果有一个是活的. 注意我们不加锁设备自旋锁, 直到定时器被删除后; 如果定时器函数在我们可删除它之前运行, 反过来做会有死锁. 在设备加锁下, 我们调用一个内核函数, 称为 check_disk_change, 来检查是否已发生一个介质改变. 可能有人争论说内核应当做这个调用, 但是标准模式是为驱动来在打开时处理它.

最后一步是递增用户计数并且返回.

释放方法的任务是, 相反, 来递减用户计数, 以及, 如果被指示了, 启动介质移出定时器:

```
static int sbull_release(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;
        spin_lock(&dev->lock)
        ;
        dev->users--;
        if (!dev->users)
        {
                dev->timer.expires = jiffies + INVALIDATE_DELAY;
                add_timer(&dev->timer);
        }

        spin_unlock(&dev->lock);
        return 0;
}
```

在一个处理真实的硬件设备的驱动中, open 和 release 方法应当相应地设置驱动和硬件的状态. 这个工作可能包括起停磁盘, 加锁一个可移出设备的门, 分配 DMA 缓冲, 等等.

你可能奇怪谁实际上打开了一个块设备. 有一些操作可导致一个块设备从用户空间直接打开; 这包括分区一个磁盘, 在一个分区上建立一个文件系统, 或者运行一个文件系统检查器. 当加载一个分区时, 块驱动也可看到一个 open 调用. 在这个情况下, 没有用户空间进程持有一个这个设备的打开的文件描述符; 相反, 打开的文件被内核自身持有. 块驱动无法知道一个加载操作(它从内核打开设备)和调用如 mkfs 工具(从用户空间打开它)之间的差别.

16.2.2. 支持可移出的介质

block_device_operations 结构包含 2 个方法来支持可移出介质. 如果你为一个非可移出设备编写一个驱动, 你可安全地忽略这些方法. 它们的实现是相对直接的.

media_changed 方法被调用(从 check_disk_change) 来看是否介质已经被改变; 它应当返回一个非零值, 如果已经发生. sbull 实现是简单的; 它查询一个已被设置的标志, 如果介质移出定时器已超时:

```
int sbull_media_changed(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;
        return dev->media_change;
}
```

revalidate 方法在介质改变后被调用; 它的工作是做任何需要的事情来准备驱动对新介质的操作, 如果有. 在调用 revalidate 之后, 内核试图重新读分区表并且启动这个设备. sbull 的实现仅仅复位 media_change 标志并且清零设备内存来模拟一个空盘插入.

```
int sbull_revalidate(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;

        if (dev->media_change)
        {
                dev->media_change = 0;
                memset (dev->data, 0, dev->size);
        }
        return 0;
}
```

16.2.3. ioctl 方法

块设备可提供一个 ioctl 方法来进行设备控制函数. 高层的块子系统代码在你的驱动能见到它们之前解释许多的 ioctl 命令, 但是(全部内容见 drivers/block/ioctl.c , 在内核源码中). 实际上, 一个现代的块驱动根本不必实现许多的 ioctl 命令.

sbull ioctl 方法只处理一个命令 -- 一个对设备的结构的请求:

```
int sbull_ioctl (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg)
{
        long size;
```

```
        struct hd_geometry geo;
        struct sbull_dev *dev = filp->private_data;

        switch(cmd)
        {
        case HDIO_GETGEO:
                /*
                * Get geometry: since we are a virtual device, we have to make
                * up something plausible. So we claim 16 sectors, four heads,
                * and calculate the corresponding number of cylinders. We set the
                * start of data at sector four.
                */
                size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
                geo.cylinders = (size & ~0x3f) >> 6;
                geo.heads = 4;
                geo.sectors = 16;
                geo.start = 4;
                if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
                        return -EFAULT;
                return 0;
        }

        return -ENOTTY; /* unknown command */
}
```

提供排列信息可能看来象一个奇怪的任务，因为我们的设备是纯粹虚拟的并且和磁道和柱面没任何关系. 甚至大部分真正的块硬件都已很多年不再有很多更复杂的结构. 内核不关心一个块设备的排列；只把它看作一个扇区的线性数组. 但是，有某些用户工具仍然想能够查询一个磁盘的排列. 特别的，fdisk 工具，它编辑分区表，依靠柱面信息并且如果这个信息没有则不能正确工作.

我们希望 sbull 设备是可分区的，即便使用老的，简单的工具. 因此，我们已提供了一个 ioctl 方法，这个方法提供了一个可靠的能够匹配我们设备容量的排列的假象. 大部分磁盘驱动做类似的事情. 注意，象通常，扇区计数被转换，如果需要，来匹配内核使用的 512-字节 的惯例.

16.3. 请求处理

每个块驱动的核心是它的请求函数. 这个函数是真正做工作的地方 --或者至少开始的地方；剩下的都是开销. 因此，我们花不少时间来看在块驱动中的请求处理.

一个磁盘驱动的性能可能是系统整个性能的关键部分. 因此，内核的块子系统编写时在性能上考虑了很多；它做所有可能的事情来使你的驱动从它控制的设备上获得最多. 这是一个好事情，其中它盲目地使能快速 I/O. 另一方面，块子系统没必要在驱动 API 中曝露大量复杂性. 有可能编写一个非常简单的请求函数(我们将很快见到)，但是如果你的驱动必须在一个高层次上操作复杂的硬件，它可能是任何样子.

16.3.1. 对请求方法的介绍

块驱动的请求方法有下面的原型:

```
void request(request_queue_t *queue);
```

这个函数被调用, 无论何时内核认为你的驱动是时候处理对设备的读, 写, 或者其他操作. 请求函数在返回之前实际不需要完成所有的在队列中的请求; 实际上, 它可能不完成它们任何一个, 对大部分真实设备. 它必须, 但是, 驱动这些请求并且确保它们最终被驱动全部处理.

每个设备有一个请求队列. 这是因为实际的从和到磁盘的传输可能在远离内核请求它们时发生, 并且因为内核需要这个灵活性来调度每个传送, 在最好的时刻(将影响磁盘上邻近扇区的请求集合到一起, 例如). 并且这个请求函数, 你可能记得, 和一个请求队列相关, 当这个队列被创建时. 让我们回顾 sbull 如何创建它的队列:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

这样, 当这个队列被创建时, 请求函数和它关联到一起. 我们还提供了一个自旋锁作为队列创建过程的一部分. 无论何时我们的请求函数被调用, 内核持有这个锁. 结果, 请求函数在原子上下文中运行; 它必须遵循所有的 5 章讨论过的原子代码的通用规则.

在你的请求函数持有锁时, 队列锁还阻止内核去排队任何对你的设备的其他请求. 在一些条件下, 你可能考虑在请求函数运行时丢弃这个锁. 如果你这样做, 但是, 你必须保证不存取请求队列, 或者任何其他的被这个锁保护的数据结构, 在这个锁不被持有时. 你必须重新请求这个锁, 在请求函数返回之前.

最后, 请求函数的启动(常常地)与任何用户空间进程之间是完全异步的. 你不能假设内核运行在发起当前请求的进程上下文. 你不知道由这个请求提供的 I/O 缓冲是否在内核或者用户空间. 因此任何类型的明确存取用户空间的操作都是错误的并且将肯定引起麻烦. 如你将见到的, 你的驱动需要知道的关于请求的所有事情, 都包含在通过请求队列传递给你的结构中.

16.3.2. 一个简单的请求方法

sbull 例子驱动提供了几个不同的方法给请求处理. 缺省地, sbull 使用一个方法, 称为 sbull_request, 它打算作为一个最简单地请求方法的例子. 别忙, 它在这里:

```
static void sbull_request(request_queue_t *q)
{
        struct request *req;
        while ((req = elv_next_request(q)) != NULL) {
                struct sbull_dev *dev = req->rq_disk->private_data;
                if (! blk_fs_request(req)) {

                        printk (KERN_NOTICE "Skip non-fs request\n");
                        end_request(req, 0);
                        continue;
                }
                sbull_transfer(dev, req->sector, req->current_nr_sectors,
                               req->buffer, rq_data_dir(req));
```

```
            end_request(req, 1);
        }
}
```

这个函数介绍了 struct request 结构. 我们之后将详细检查 struct request; 现在, 只需说它表示一个我们要执行的块 I/O 请求.

内核提供函数 elv_next_request 来获得队列中第一个未完成的请求; 当没有请求要被处理时这个函数返回 NULL. 注意 elf_next 不从队列里去除请求. 如果你连续调用它 2 次, 它 2 次都返回同一个请求结构. 在这个简单的操作模式中, 请求只在它们完成时被剥离队列.

一个块请求队列可包含实际上不从磁盘和自磁盘移动块的请求. 这些请求可包括供应商特定的, 低层的诊断操作或者和特殊设备模式相关的指令, 例如给可记录介质的报文写模式. 大部分块驱动不知道如何处理这样的请求, 并且简单地失败它们; sbull 也以这种方式工作. 对 block_fs_request 的调用告诉我们是否我们在查看一个文件系统请求--一个一旦数据块的. 如果这个请求不是一个文件系统请求, 我们传递它到 end_request:

```
void end_request(struct request *req, int succeeded);
```

当我们处理了非文件系统请求, 之后我们传递 succeeded 为 0 来指示我们没有成功完成这个请求. 否则, 我们调用 sbull_transfer 来真正移动数据, 使用一套在请求结构中提供的成员:

sector_t sector;

　　我们设备上起始扇区的索引. 记住这个扇区号, 象所有这样的在内核和驱动之间传递的数目, 是以 512-字节扇区来表示的. 如果你的硬件使用一个不同的扇区大小, 你需要相应地调整扇区. 例如, 如果硬件是 2048-字节的扇区, 你需要用 4 来除起始扇区号, 在安放它到对硬件的请求之前.

unsigned long nr_sectors;

　　要被传送的扇区(512-字节)数目.

char *buffer;

　　一个指向缓冲的指针, 数据应当被传送到或者从的缓冲. 这个指针是一个内核虚拟地址并且可被驱动直接解引用, 如果需要.

rq_data_dir(struct request *req);

　　这个宏从请求中抽取传送的方向; 一个 0 返回值表示从设备中读, 非 0 返回值表示写入设备.

有了这个信息, sbull 驱动可实现实际的数据传送, 使用一个简单的 memcpy 调用 -- 我们数据已经在内存, 毕竟. 进行这个拷贝操作的函数(sbull_transfer) 也处理扇区大小的调整, 并确保我们没有拷贝超过我们的虚拟设备的尾.

```
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector, unsigned long
nsect, char *buffer, int write)
{
        unsigned long offset = sector*KERNEL_SECTOR_SIZE;
        unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
        if ((offset + nbytes) > dev->size)
        {
                printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
                return;
        }
        if (write)
                memcpy(dev->data + offset, buffer, nbytes);
        else
                memcpy(buffer, dev->data + offset, nbytes);
}
```

用这个代码, sbull 实现了一个完整的, 简单的基于 RAM 的磁盘设备. 但是, 对于很多类型的设备, 它不是一个实际的驱动, 由于几个理由.

这些原因的第一个是 sbull 同步执行请求, 一次一个. 高性能的磁盘设备能够在同时有很多个请求停留; 磁盘的板上控制器因此可以优化的顺序(有人希望)执行它们. 如果我们只处理队列中的第一个请求, 我们在给定时间不能有多个请求被满足. 能够工作于多个请求要求对请求队列和请求结构的深入理解; 下面几节会帮助来建立这种理解.

但是, 有另外一个问题要考虑. 当系统进行大的传输, 包含多个在一起的磁盘扇区, 就获得最好的性能. 磁盘操作的最高开销常常是读写头的定位; 一旦这个完成, 实际上需要的读或者写数据的时间几乎可忽略. 设计和实现文件系统和虚拟内存子系统的开发者理解这点, 因此他们尽力在磁盘上连续地查找相关的数据, 并且在一次请求中传送尽可能多扇区. 块子系统也在这个方面起作用; 请求队列包含大量逻辑,目的是找到邻近的请求并且接合它们为更大的操作.

sbull 驱动, 但是, 采取所有这些工作并且简单地忽略它. 一次只有一个缓冲被传送, 意味着最大的单次传送几乎从不超过单个页的大小. 一个块驱动能做的比那个要好的多, 但是它需要一个对请求结构和 bio 结构的更深的理解, 请求是从它们建立的.

下面几节更深入地研究块层如何完成它的工作, 已经这些工作导致的数据结构.

16.3.3. 请求队列

最简单的说, 一个块请求队列就是: 一个块 I/O 请求的队列. 如果你往下查看, 一个请求队列是一令人吃惊得复杂的数据结构. 幸运的是, 驱动不必担心大部分的复杂性.

请求队列跟踪等候的块 I/O 请求. 但是它们也在这些请求的创建中扮演重要角色. 请求队列存储参数, 来描述这个设备能够支持什么类型的请求: 它们的最大大小, 多少不同的段可进入一个请求, 硬件扇区大小, 对齐要求, 等等. 如果你的请求队列被正确配置了, 它应当从不交给你一个你的设备不能处理的请求.

请求队列还实现一个插入接口, 这个接口允许使用多 I/O 调度器(或者电梯). 一个 I/O 调度器的工作是提交 I/O 请求给你的驱动, 以最大化性能的方式. 为此, 大部分 I/O 调

度器累积批量的 I/O 请求，排列它们为递增(或递减)的块索引顺序，并且以那个顺序提交请求给驱动。磁头，当给定一列排序的请求时，从磁盘的一头到另一头工作，非常象一个满载的电梯，在一个方向移动直到所有它的"请求"(等待出去的人)已被满足。2.6 内核包含一个"底线调度器"，它努力确保每个请求在预设的最大时间内被满足，以及一个"预测调度器"，它实际上短暂停止设备，在一个预想中的读请求之后，这样另一个邻近的读将几乎是马上到达。到本书为止，缺省的调度器是预测调度器，它看来有最好的交互的系统性能。

I/O 调度器还负责合并邻近的请求。当一个新 I/O 请求被提交给调度器，它在队列里搜寻包含邻近扇区的请求；如果找到一个，并且如果结果的请求不是太大，这 2 个请求被合并。

请求队列有一个 struct request_queue 或者 request_queue_t 类型。这个类型，和许多操作它的函数，定义在 <linux/blkdev.h>。如果你对请求队列的实现感兴趣，你可找到大部分代码在 drivers/block/ll_rw_block.c 和 elevator.c。

16.3.3.1. 队列的创建和删除

如同我们在我们的例子代码中见到的，一个请求队列是一个动态的数据结构，它必须被块 I/O 子系统创建。这个创建和初始化一个队列的函数是：

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock);
```

当然，参数是，这个队列的请求函数和一个控制对队列存取的自旋锁。这个函数分配内存(实际上，不少内存)并且可能失败因为这个；你应当一直检查返回值，在试图使用这个队列之前。

作为初始化一个请求队列的一部分，你可设置成员 queuedata(它是一个 void * 指针)为任何你喜欢的值。这个成员是请求队列的对于我们在其他结构中见到的 private_data 的对等体。

为返回一个请求队列给系统(在模块卸载时间，通常)，调用 blk_cleanup_queue：

```
void blk_cleanup_queue(request_queue_t *);
```

这个调用后，你的驱动从给定的队列中不再看到请求,并且不应当再次引用它。

16.3.3.2. 排队函数

有非常少的函数来操作队列中的请求 -- 至少，考虑到驱动。你必须持有队列锁，在你调用这些函数之前。

返回要处理的下一个请求的函数是 elv_next_request：

```
struct request *elv_next_request(request_queue_t *queue);
```

我们已经在简单的 sbull 例子中见到这个函数. 它返回一个指向下一个要处理的请求的指针(由 I/O 调度器所决定的)或者 NULL 如果没有请求要处理. elv_next_request 留这个请求在队列上, 但是标识它为活动的; 这个标识阻止了 I/O 调度器试图合并其他的请求到这些你开始执行的.

为实际上从一个队列中去除一个请求, 使用 blkdev_dequeue_request:

```
void blkdev_dequeue_request(struct request *req);
```

如果你的驱动同时从同一个队列中操作多个请求, 它必须以这样的方式将它们解出队列.

如果你由于同样的理由需要放置一个出列请求回到队列中, 你可以调用:

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

16.3.3.3. 队列控制函数

块层输出了一套函数, 可被驱动用来控制一个请求队列如何操作. 这些函数包括:

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果你的设备已到到达一个状态, 它不能处理等候的命令, 你可调用 blk_stop_queue 来告知块层. 在这个调用之后, 你的请求函数将不被调用直到你调用 blk_start_queue. 不用说, 你不应当忘记重启队列, 当你的设备可处理更多请求时. 队列锁必须被持有当调用任何一个这些函数时.

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

告知内核你的设备可进行 DMA 的最高物理地址的函数. 如果一个请求包含一个超出这个限制的内存引用, 一个反弹缓冲将被用来给这个操作; 当然, 这是一个进行块 I/O 的昂贵方式, 并且应当尽量避免. 你可在这个参数中提供任何可能的值, 或者使用预先定义的符号 BLK_BOUNCE_HIGH(使用反弹缓冲给高内存页), BLK_BOUNCE_ISA (驱动只可 DMA 到 16MB 的 ISA 区), 或者 BLK_BOUCE_ANY(驱动可进行 DMA 到任何地址). 缺省值是 BLK_BOUNCE_HIGH.

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

设置参数的函数, 这些参数描述可被设备满足的请求. blk_queue_max 可用来以扇区方式设置任一请求的最大的大小; 缺省是 255. blk_queue_max_phys_segments 和 blk_queue_max_hw_segments 都控制多少物理段(系统内存中不相邻的区)可包含在一个请求中. 使用 blk_queue_max_phys_segments 来说你的驱动准备处理多少段; 例如, 这可能是一个静态分配的散布表的大小. blk_queue_max_hw_segments, 相反, 是设备可处理的最多的段数. 这 2 个参数缺

省都是 128. 最后，blk_queue_max_segment_size 告知内核任一个请求的段可能是多大字节；缺省是 65,536 字节.

blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);

一些设备无法处理跨越一个特殊大小内存边界的请求；如果你的设备是其中之一，使用这个函数来告知内核这个边界. 例如，如果你的设备处理跨 4-MB 边界的请求有困难，传递一个 0x3fffff 掩码. 缺省的掩码是 0xffffffff.

void blk_queue_dma_alignment(request_queue_t *queue, int mask);

告知内核关于你的设备施加于 DMA 传送的内存对齐限制的函数. 所有的请求被创建有给定的对齐，并且请求的长度也匹配这个对齐. 缺省的掩码是 0x1ff，它导致所有的请求被对齐到 512-字节边界.

void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);

告知内核你的设备的硬件扇区大小. 所有由内核产生的请求是这个大小的倍数并且被正确对齐. 所有的在块层和驱动之间的通讯继续以 512-字节扇区来表达，但是.

16.3.4. 请求的分析

在我们的简单例子里，我们遇到了这个请求结构. 但是，我们未曾接触这个复杂的数据结构. 在本节，我们看，详细地，块 I/O 请求在 Linux 内核中如何被表示.

每个请求结构代表一个块 I/O 请求，尽管它可能是由几个独立的请求在更高层次合并而成. 对任何特殊的请求而传送的扇区可能分布在整个主内存，尽管它们常常对应块设备中的多个连续的扇区. 这个请求被表示为多个段，每个对应一个内存中的缓冲. 内核可能合并多个涉及磁盘上邻近扇区的请求，但是它从不合并在单个请求结构中的读和写操作. 内核还确保不合并请求，如果结果会破坏任何的在前面章节中描述的请求队列限制.

基本上，一个请求结构被实现为一个 bio 结构的链表，结合一些维护信息来使驱动可以跟踪它的位置，当它在完成这个请求中. 这个 bio 结构是一个块 I/O 请求移植的低级描述；我们现在看看它.

16.3.4.1. bio 结构

当内核，以一个文件系统的形式，虚拟文件子系统，或者一个系统调用，决定一组块必须传送到或从一个块 I/O 设备；它装配一个 bio 结构来描述那个操作. 那个结构接着被递给这个块 I/O 代码，这个代码合并它到一个存在的请求结构，或者，如果需要，创建一个新的. 这个 bio 结构包含一个块驱动需要来进行请求的任何东西，而不必涉及使这个请求启动的用户空间进程.

bio 结构，在 <linux/bio.h> 中定义，包含许多成员对驱动作者是有用的:

sector_t bi_sector;

这个 bio 要被传送的第一个(512 字节)扇区.

unsigned int bi_size;

被传送的数据大小, 以字节计. 相反, 常常更易使用 bio_sectors(bio), 一个给定以扇区计的大小的宏.

unsigned long bi_flags;

一组描述 bio 的标志; 最低有效位被置位如果这是一个写请求(尽管宏 bio_data_dir(bio)应当用来代替直接加锁这个标志).

unsigned short bio_phys_segments;
unsigned short bio_hw_segments;

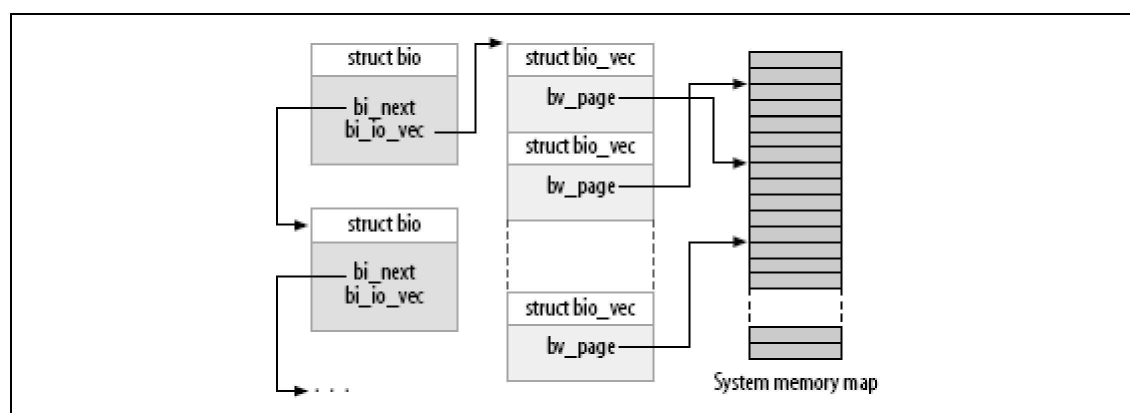
包含在这个 BIO 中的物理段的数目, 和在 DMA 映射完成后被硬件看到的段数目, 分别地.

一个 bio 的核心, 但是, 是一个称为 bi_io_vec 的数组, 它由下列结构组成:

```
struct bio_vec {
 struct page  *bv_page;
 unsigned int  bv_len;
 unsigned int  bv_offset;
};
```

图 bio 结构显示了这些结构如何结合在一起. 如同你所见到的, 在一个块 I/O 请求被转换为一个 bio 结构后, 它已被分为单独的物理内存页. 所有的一个驱动需要做的事情是步进全部这个结构数组(它们有 bi_vcnt 个), 和在每个页内传递数据(但是只 len 字节, 从 offset 开始).

图 16.1. bio 结构



直接使用 bi_io_vec 数组不被推荐, 为了内核开发者可以在以后改变 bio 结构而不会引起破坏. 为此, 一组宏被提供来简化使用 bio 结构. 开始的地方是 bio_for_each_segment, 它简单地循环 bi_io_vec 数组中每个未被处理的项. 这个宏应当如下用:

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
 /* Do something with this segment
}
```

在这个循环中, bvec 指向当前的 bio_vec 项, 并且 segno 是当前的段号. 这些值可被用来设置 DMA 发送器(一个使用 blk_rq_map_sg 的替代方法在"块请求和 DMA"一节中描述). 如果你需要直接存取页, 你应当首先确保一个正确的内核虚拟地址存在; 为此, 你可使用:

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

这个底层的函数允许你直接映射在一个给定的 bio_vec 中找到的缓冲, 由索引 i 所指定的. 一个原子的 kmap 被创建; 调用者必须提供合适的来使用的槽位(如同在 15 章的"内存映射和 struct page"一节中描述的).

块层还维护一组位于 bio 结构的指针来跟踪请求处理的当前状态. 几个宏来提供对这个状态的存取:

`struct page *bio_page(struct bio *bio);`

返回一个指向页结构的指针, 表示下一个被传送的页.

`int bio_offset(struct bio *bio);`

返回页内的被传送的数据的偏移.

`int bio_cur_sectors(struct bio *bio);`

返回要被传送出当前页的扇区数.

`char *bio_data(struct bio *bio);`

返回一个内核逻辑地址, 指向被传送的数据. 注意这个地址可用仅当请求的页不在高内存中; 在其他情况下调用它是一个错误. 缺省地, 块子系统不传递高内存缓冲到你的驱动, 但是如果你已使用 blk_queue_bounce_limit 改变设置, 你可能不该使用 bio_data.

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

bio_kmap_irq 给任何缓冲返回一个内核虚拟地址, 不管它是否在高或低内存. 一个原子 kmap 被使用, 因此你的驱动在这个映射被激活时不能睡眠. 使用 bio_kunmap_irq 来去映射缓冲. 注意因为使用一个原子 kmap, 你不能一次映射多于一个段.

刚刚描述的所有函数都存取当前缓冲 -- 还未被传送的第一个缓冲，只要内核知道. 驱动常常想使用 bio 中的几个缓冲，在它们任何一个指出完成之前(使用 end_that_request_first，马上就讲到)，因此这些函数常常没有用. 几个其他的宏存在来使用 bio 结构的内部接口(详情见 <linux/bio.h>).

16.3.4.2. 请求结构成员

现在我们有了 bio 结构如何工作的概念，我们可以深入 struct request 并且看请求处理如何工作. 这个结构的成员包括:

sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;

追踪请求硬件完成的扇区的成员. 第一个尚未被传送的扇区被存储到 hard_sector，已经传送的扇区总数在 hard_nr_sectors，并且在当前 bio 中剩余的扇区数是 hard_cur_sectors. 这些成员打算只用在块子系统；驱动不应当使用它们.

struct bio *bio;

bio 是给这个请求的 bio 结构的链表. 你不应当直接存取这个成员；使用 rq_for_each_bio(后面描述) 代替.

char *buffer;

在本章前面的简单驱动例子使用这个成员来找到传送的缓冲. 随着我们的深入理解，我们现在可见到这个成员仅仅是在当前 bio 上调用 bio_data 的结果.

unsigned short nr_phys_segments;

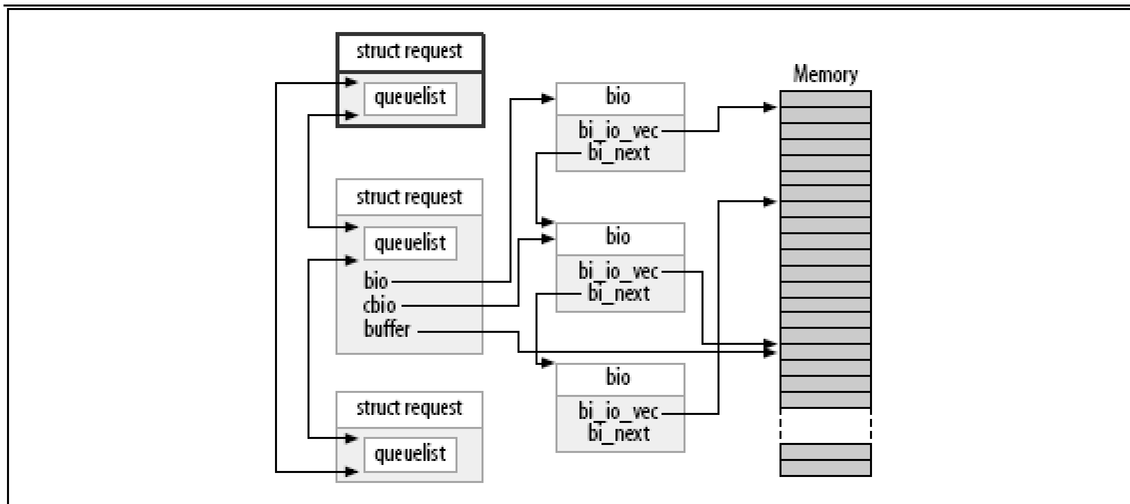
被这个请求在物理内存中占用的独特段的数目，在邻近页已被合并后.

struct list_head queuelist;

链表结构(如同在 11 章中"链表"一节中描述的)，连接这个请求到请求队列. 如果(并且只是)你从队列中去除 blkdev_dequeue_request，你可能使用这个列表头来跟踪这个请求，在一个被你的驱动维护的内部列表中.

图 一个带有一个部分被处理的请求的请求队列 展示了请求队列和它的组件 bio 结构如何对应到一起. 在图中，这个请求已被部分满足. cbio 和 buffer 处于指向尚未传送的第一个 bio.

图 16.2. 一个带有一个部分被处理的请求的请求队列



有许多不同的字段在请求结构中，但是本节中的列表应当对大部分驱动编写者是足够的.

16.3.4.3. 屏障请求

块层在你的驱动见到它们之前重新排序来提高 I/O 性能. 你的驱动，也可以重新排序请求，如果有理由这样做. 常常地，这种重新排序通过传递多个请求到驱动并且使硬件考虑优化的顺序来实现. 但是，对于不严格的请求顺序有一个问题：有些应用程序要求保证某些操作在其他的启动前完成. 例如，关系数据库管理者，必须绝对确保它们的日志信息刷新到驱动器，在执行在数据库内容上的一次交易之前. 日志式文件系统，现在在大部分 Linux 系统中使用，有非常类似的排序限制. 如果错误的操作被重新排序，结果可能是严重的，无法探测的数据破坏.

2.6 块层解决这个问题通过一个屏障请求的概念. 如果一个请求被标识为 REQ_HARDBARRER 标志，它必须被写入驱动器在任何后续的请求被初始化之前. "被写入设备"，我们意思是数据必须实际位于并且是持久的在物理介质中. 许多的驱动器进行写请求的缓存；这个缓存提高了性能，但是它可能使屏障请求的目的失败. 如果一个电力失效在关键数据仍然在驱动器的缓存中时发生，数据仍然被丢失即便驱动器报告完成. 因此一个实现屏障请求的驱动器必须采取步骤来强制驱动器真正写这些数据到介质中.

如果你的驱动器尊敬屏障请求，第一步是通知块层这个事实. 屏障处理是另一个请求队列；它被设置为：

```
void blk_queue_ordered(request_queue_t *queue, int flag);
```

为指示你的驱动实现了屏障请求，设置 flag 参数为一个非零值.

实际的屏障请求实现是简单地测试在请求结构中关联的标志. 已经提供了一个宏来进行这个测试：

```
int blk_barrier_rq(struct request *req);
```

如果这个宏返回一个非零值，这个请求是一个屏障请求. 根据你的硬件如何工作，你可能必须停止从队列中获取请求，直到屏障请求已经完成. 另外的驱动器能理解屏障请求；在这个情况中，你的驱动所有的必须做的是对这些驱动器发出正确的操作.

16.3.4.4. 不可重入请求

块驱动常常试图重试第一次失败的请求. 这个做法可产生一个更加可靠的系统并且帮助来避免数据丢失. 内核，但是，有时标识请求为不可重入的. 这样的请求应当完全尽快失败，如果它们无法在第一次试的时候执行.

如果你的驱动在考虑重试一个失败的请求，他应当首先调用:

```
int blk_noretry_request(struct request *req);
```

如果这个宏返回非零值，你的驱动应当放弃这个请求，使用一个错误码来代替重试它.

16.3.5. 请求完成函数

如同我们将见到的，有几个不同的方式来使用一个请求结构. 它们所有的都使用几个通用的函数，但是，它们处理一个 I/O 请求或者部分请求的完成. 这 2 个函数都是原子的并且可从一个原子上下文被安全地调用.

当你的设备已经完成传送一些或者全部扇区，在一个 I/O 请求中，它必须通知块子系统，使用:

```
int end_that_request_first(struct request *req, int success, int count);
```

这个函数告知块代码，你的驱动已经完成 count 个扇区地传送，从你最后留下的地方开始. 如果 I/O 是成功的，传递 success 为 1；否则传递 0. 注意你必须指出完成，按照从第一个扇区到最后一个的顺序；如果你的驱动和设备有些共谋来乱序完成请求，你必须存储这个乱序的完成状态直到介入的扇区已经被传递.

从 end_that_request_first 的返回值是一个指示，指示是否所有的这个请求中的扇区已经被传送或者没有. 一个 0 返回值表示所有的扇区已经被传送并且这个请求完成. 在这点，你必须使用 blkdev_dequeue_request 来从队列中解除请求(如果你还没有这样做)并且传递它到:

```
void end_that_request_last(struct request *req);
```

end_that_request_last 通知任何在等待这个请求的人，这个请求已经完成并且回收这个请求结构；它必须在持有队列锁时被调用.

在我们的简单的 sbull 例子里，我们不使用任何上面的函数. 相反，那个例子，被称为 end_request. 为显示这个调用的效果，这里有整个的 end_request 函数，如果在 2.6.10 内核中见到的:

```
void end_request(struct request *req, int uptodate)
```

```
{
 if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
 add_disk_randomness(req->rq_disk);
 blkdev_dequeue_request(req);
 end_that_request_last(req);
 }
}
```

函数 add_disk_randomness 使用块 I/O 请求的定时来贡献熵给系统的随机数池; 它应当被调用仅当磁盘的定时是真正的随机的. 对大部分的机械设备这是真的, 但是对一个基于内存的虚拟设备它不是真的, 例如 sbull. 因此, 下一节中更复杂的 sbull 版本不调用 add_disk_randomness.

16.3.5.1. 使用 bio

现在你了解了足够多的来编写一个块驱动, 可直接使用组成一个请求的 bio 结构. 但是, 一个例子可能会有帮助. 如果这个 sbull 驱动被加载为 request_mode 参数被设为 1, 它注册一个知道 bio 的请求函数来代替我们上面见到的简单函数. 那个函数看来如此:

```
static void sbull_full_request(request_queue_t *q)
{
        struct request *req;
        int sectors_xferred;
        struct sbull_dev *dev = q->queuedata;
        while ((req = elv_next_request(q)) != NULL) {
                if (! blk_fs_request(req)) {
                        printk (KERN_NOTICE "Skip non-fs request\n");

                        end_request(req, 0);
                        continue;
                }
                sectors_xferred = sbull_xfer_request(dev, req);
                if (! end_that_request_first(req, 1, sectors_xferred)) {
                        blkdev_dequeue_request(req);
                        end_that_request_last(req);
                }
        }
}
```

这个函数简单地获取每个请求, 传递它到 sbull_xfer_request, 接着使用 end_that_request_first 和, 如果需要, end_that_request_last 来完成它. 因此, 这个函数在处理高级队列并且请求管理部分问题. 真正执行一个请求的工作, 但是, 落入 sbull_xfer_request:

```
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
        struct bio *bio;
        int nsect = 0;

        rq_for_each_bio(bio, req)
```

```
        {
                sbull_xfer_bio(dev, bio);
                nsect += bio->bi_size/KERNEL_SECTOR_SIZE;

        }
        return nsect;
}
```

这里我们介绍另一个宏: rq_for_each_bio. 如同你可能期望的, 这个宏简单地步入请求中的每个 bio 结构, 给我们一个可传递给 sbull_xfer_bio 用于传输的指针. 那个函数看来如此:

```
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
        int i;
        struct bio_vec *bvec;
        sector_t sector = bio->bi_sector;

        /* Do each segment independently. */
        bio_for_each_segment(bvec, bio, i)
        {
                char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
                sbull_transfer(dev, sector, bio_cur_sectors(bio),

                               buffer, bio_data_dir(bio) == WRITE);
                sector += bio_cur_sectors(bio);
                __bio_kunmap_atomic(bio, KM_USER0);

        }
        return 0; /* Always "succeed" */
}
```

这个函数简单地步入每个 bio 结构中的段, 获得一个内核虚拟地址来存取缓冲, 接着调用之前我们见到的同样的 sbull_transfer 函数来拷贝数据.

每个设备有它自己的需要, 但是, 作为一个通用的规则, 刚刚展示的代码应当作为一个模型, 给许多的需要深入 bio 结构的情形.

16.3.5.2. 块请求和 DMA

如果你工作在一个高性能块驱动上, 你有机会使用 DMA 来进行真正的数据传输. 一个块驱动当然可步入 bio 结构, 如同上面描述的, 为每一个创建一个 DMA 映射, 并且传递结构给设备. 但是, 有一个更容易的方法, 如果你的驱动可进行发散/汇聚 I/O. 函数:

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct scatterlist *list);
```

使用来自给定请求的全部段填充给定的列表. 内存中邻近的段在插入散布表之前被接合, 因此你不需要自己探测它们. 返回值是列表中的项数. 这个函数还回传, 在它第 3 个参数, 一个适合传递给 dma_map_sg 的散布表.(关于 dma_map_sg 的更多信息见 15 章的"发散-汇聚映射"一节).

你的驱动必须在调用 blk_rq_map_sg 之前给散布表分配存储. 这个列表必须能够至少持有这个请求有的物理段那么多的项; struct request 成员 nr_phys_segments 持有那个数量, 它不能超过由 blk_queue_max_phys_segments 指定的物理段的最大数目.

如果你不想 blk_rq_map_sg 来接合邻近的段, 你可改变这个缺省的行为, 使用一个调用诸如:

```
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

一些 SCSI 磁盘驱动用这样的方式标识它们的请求队列, 因为它们没有从接合请求中获益.

16.3.5.3. 不用一个请求队列

前面, 我们已经讨论了内核所作的在队列中优化请求顺序的工作; 这个工作包括排列请求和, 或许, 甚至延迟队列来允许一个预期的请求到达. 这些技术在处理一个真正的旋转的磁盘驱动器时有助于系统的性能. 但是, 使用一个象 sbull 的设备它们是完全浪费了. 许多面向块的设备, 例如闪存阵列, 用于数字相机的存储卡的读取器, 并且 RAM 盘真正地有随机存取的性能, 包含从高级的请求队列逻辑中获益. 其他设备, 例如软件 RAID 阵列或者被逻辑卷管理者创建的虚拟磁盘, 没有这个块层的请求队列被优化的性能特征. 对于这类设备, 它最好直接从块层接收请求, 并且根本不去烦请求队列.

对于这些情况, 块层支持"无队列"的操作模式. 为使用这个模式, 你的驱动必须提供一个"制作请求"函数, 而不是一个请求函数. make_request 函数有这个原型:

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

注意一个请求队列仍然存在, 即便它从不会真正有任何请求. make_request 函数用一个 bio 结构作为它的主要参数, 这个 bio 结构表示一个或多个要传送的缓冲. make_request 函数做 2 个事情之一: 它可或者直接进行传输, 或者重定向这个请求到另一个设备.

直接进行传送只是使用我们前面描述的存取者方法来完成这个 bio. 因为没有使用请求结构, 但是, 你的函数应当通知这个 bio 结构的创建者直接指出完成, 使用对 bio_endio 的调用:

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

这里, bytes 是你至今已经传送的字节数. 它可小于由这个 bio 整体所代表的字节数; 在这个方式中, 你可指示部分完成, 并且更新在 bio 中的内部的"当前缓冲"指针. 你应当再次调用 bio_endio 在你的设备进行进一步处理时, 或者当你不能完成这个请求指出一个错误. 错误是通过提供一个非零值给 error 参数来指示的; 这个值通常是一个错误码, 例如 -EIO. make_request 应当返回 0, 不管这个 I/O 是否成功.

如果 sbull 用 request_mode=2 加载, 它操作一个 make_request 函数. 因为 sbull 已经有一个函数看传送单个 bio, 这个 make_request 函数简单:

```
static int sbull_make_request(request_queue_t *q, struct bio *bio)
```

```
{
        struct sbull_dev *dev = q->queuedata;
        int status;
        status = sbull_xfer_bio(dev, bio);
        bio_endio(bio, bio->bi_size, status);
        return 0;
}
```

请注意你应当从不调用 bio_endio 从一个通常的请求函数；那个工作由 end_that_request_first 代替来处理.

一些块驱动，例如那些实现卷管理者和软件 RAID 阵列的，真正需要重定向请求到另一个设备来处理真正的 I/O. 编写这样的一个驱动超出了本书的范围. 我们，但是，注意如果 make_request 函数返回一个非零值，bio 被再次提交. 一个"堆叠"驱动，可，因此，修改 bi_bdev 成员来指向一个不同的设备，改变起始扇区值，接着返回；块系统接着传递 bio 到新设备. 还有一个 bio_split 调用来划分一个 bio 到多个块以提交给多个设备. 尽管如果队列参数被之前设置，划分一个 bio 几乎从不需要.

任何一个方式，你都必须告知块子系统，你的驱动在使用一个自定义的 make_request 函数. 为此，你必须分配一个请求队列，使用：

```
request_queue_t *blk_alloc_queue(int flags);
```

这个函数不同于 blk_init_queue，它不真正建立队列来持有请求. flags 参数是一组分配标志被用来为队列分配内存；常常地正确值是 GFP_KERNEL. 一旦你有一个队列，传递它和你的 make_request 函数到 blk_queue_make_request：

```
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

sbull 代码来设置 make_request 函数，象：

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue == NULL)
        goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

对于好奇的人，花些时间深入 drivers/block/ll_rw_block.c 会发现，所有的队列都有一个 make_request 函数. 缺省的版本，generic_make_request，处理 bio 和一个请求结构的结合. 通过提供一个它自己的 make_request 函数，一个驱动真正只覆盖一个特定的请求队列方法，并且排序大部分工作.

16.4. 一些其他的细节

本节涵盖块层的几个其他的方面，对于高级读者可能有兴趣. 对于编写一个正确的驱动下面的内容都不需要，但是它们在某些情况下可能是有用的.

16.4.1. 命令预准备

块层为驱动提供一个进制来检查和预处理请求, 在它们被从 elv_next_request 返回前. 这个机制允许驱动提前设立真正的驱动器命令, 决定是否这个请求可被完全处理, 或者进行其他的维护工作.

如果你想使用这个特性, 创建一个命令准备函数, 它要适应这个原型:

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
```

请求结构包含一个成员 cmd, 它是一个 BLK_MAX_CDB 字节的数组; 这个数组可被这个准备函数用来存储实际的硬件命令(或者任何其他的有用信息). 这个函数应当返回一个下列的值:

BLKPREP_OK

命令准备正常进行, 并且这个请求可被传递给你的驱动的请求函数.

BLKPREP_KILL

这个请求不能完成; 它带有一个错误码而失败.

BLKPREP_DEFER

这个请求这次无法完成. 它位于队列的前面, 但是不能传递给请求函数.

准备函数被 elv_next_request 在请求返回到你的驱动之前立刻调用. 如果这个函数返回 BLKPREP_DEFER, 从 elv_next_request 返回给你的驱动的返回值是 NULL. 这个操作描述可能是有用的, 如果, 例如你的设备已达到它能够等候的请求的最大数目.

为使块层调用你的准备函数, 传递它到:

```
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

缺省地, 请求队列没有准备函数.

16.4.2. 被标识的命令排队

可同时有多个请求被激活的硬件, 常常支持某种被标识的命令排队(TCQ). TCQ 简单地说是关联一个整数 "tag" 到每个请求的技术, 注意当驱动器完成每个请求时, 他可告知驱动是哪一个. 在以前的内核版本, 实现 TCQ 的块驱动不得不自己做所有的工作; 在 2.6, 一个 TCQ 支持框架已经被添加到块层, 以给所有的驱动来使用.

如果你的驱动器进行标记命令排队, 你应当在初始化时通知内核这个事实, 使用:

```
int blk_queue_init_tags(request_queue_t *queue, int depth, struct
blk_queue_tag *tags);
```

这里, queue 是你的请求队列, 而 depth 是你的设备能够在任何时间拥有的等待的标记请求的数目. tags 是一个可选的指针指向一个 struct blk_queue_tag 结构数组; 必须

有 depth 个. 正常地, tags 可用 NULL, 并且 blk_queue_init_tags 分配这个 数组. 如果, 但是, 你需要和多个设备分享通用的 tags, 你可传递这个标记数组指针(存储在 queue_tags 成员)从另一个请求队列. 你应当从不真正自己分配这个标记数组; 块层需要初始化这个数组并且不输出这个初始化函数给模块.

因为 blk_queue_init_tags 分配内存, 它可能失败. 在那个情况下它返回一个负的错误码给调用者.

如果你的设备可处理的标记的数目改变了, 你可通知内核, 使用:

```
int blk_queue_resize_tags(request_queue_t *queue, int new_depth);
```

这个队列锁必须在这个调用期间被持有. 这个调用可能失败, 返回一个负错误码.

一个标记和一个请求结构的关联被 blk_queue_start_tag 来完成, 它必须在成员队列锁被持有时调用:

```
int blk_queue_start_tag(request_queue_t *queue, struct request *req);
```

如果一个 tag 可用, 这个函数分配它给这个请求, 存储这个标识号在 req->tag, 并且返回 0. 它还从队列中解除这个请求, 并且连接它到它自己的标识跟踪结构, 因此你的驱动应当小心不从队列中解除这个请求, 如果在使用标识. 如果没有标识可用, blk_queue_start_tag 将这个请求留在队列并且返回一个非零值.

当一个给定的请求的所有的传送都已完成, 你的驱动应当返回标识, 使用:

```
void blk_queue_end_tag(request_queue_t *queue, struct request *req);
```

再一次, 你必须持有队列锁, 在调用这个函数之前. 这个调用应当在 end_that_request_first 返回 0 之后进行(意味着这个请求完成), 但要在调用 end_that_request_last 之前. 记住这个请求已经从队列中解除, 因此它对于你的驱动在此点这样做可能是一个错误.

如果你需要找到关联到一个给定标识上的请求(当驱动器报告完成, 例如), 使用 blk_queue_find_tag:

```
struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);
```

返回值是关联的请求结构, 除非有些事情已经真的出错了.

如果事情真地出错了, 你的请求可能发现它自己不得不复位或者对其中一个它的设备进行一些其他的大动作. 在这种情况下, 任何等待中的标识命令将不会完成. 块层提供一个函数可用帮助在这种情况下恢复:

```
void blk_queue_invalidate_tags(request_queue_t *queue);
```

这个函数返回所有的等待的标识给这个池, 并且将关联的请求放回请求队列. 你调用这个函数时必须持有队列锁.

16.5. 快速参考

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
```

register_blkdev 注册一个块驱动到内核，并且，可选地，获得一个主编号. 一个驱动可被注销，使用 unregister_blkdev.

```
struct block_device_operations
```

持有大部分块驱动的方法的结构.

```
#include <linux/genhd.h>
struct gendisk;
```

描述内核中单个块设备的结构.

```
struct gendisk *alloc_disk(int minors);
void add_disk(struct gendisk *gd);
```

分配 gendisk 结构的函数，并且返回它们到系统.

```
void set_capacity(struct gendisk *gd, sector_t sectors);
```

存储设备能力(以 512-字节)在 gendisk 结构中.

```
void add_disk(struct gendisk *gd);
```

添加一个磁盘到内核. 一旦调用这个函数，你的磁盘的方法可被内核调用.

```
int check_disk_change(struct block_device *bdev);
```

一个内核函数，检查在给定磁盘驱动器中的介质改变，并且采取要求的清理动作当检测到这样一个改变.

```
#include <linux/blkdev.h>
request_queue_t blk_init_queue(request_fn_proc *request, spinlock_t *lock);
void blk_cleanup_queue(request_queue_t *);
```

处理块请求队列的创建和删除的函数.

```
struct request *elv_next_request(request_queue_t *queue);
void end_request(struct request *req, int success);
```

elv_next_request 从一个请求队列中获得下一个请求；end_request 可用在每个简单驱动器中来标识一个(或部分)请求完成.

```
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

从队列中除去一个请求，并且放回它的函数如果需要.

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果你需要阻止对你的请求函数的进一步调用，调用 blk_stop_queue 来完成. 调用 blk_start_queue 来使你的请求方法被再次调用.

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

设置各种队列参数的函数，来控制请求如何被创建给一个特殊设备；这些参数在"队列控制函数"一节中描述.

```
#include <linux/bio.h>
struct bio;
```

低级函数，表示一个块 I/O 请求的一部分.

```
bio_sectors(struct bio *bio);
bio_data_dir(struct bio *bio);
```

2 个宏定义，表示一个由 bio 结构描述的传送的大小和方向.

```
bio_for_each_segment(bvec, bio, segno);
```

一个伪控制结构，用来循环组成一个 bio 结构的各个段.

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

__bio_kmap_atomic 可用来创建一个内核虚拟地址给一个在 bio 结构中的给定的段. 映射必须使用 __bio_kunmap_atomic 来恢复.

```
struct page *bio_page(struct bio *bio);
int bio_offset(struct bio *bio);
int bio_cur_sectors(struct bio *bio);
char *bio_data(struct bio *bio);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

一组存取者宏定义，提供对一个 bio 结构中的"当前"段的存取.

```
void blk_queue_ordered(request_queue_t *queue, int flag);
int blk_barrier_rq(struct request *req);
```

如果你的驱动实现屏障请求，调用 blk_queue_ordered -- 如同它应当做的. 宏 blk_barrier_rq 返回一个非零值如果当前请求是一个屏障请求.

```
int blk_noretry_request(struct request *req);
```

这个宏返回一个非零值，如果给定的请求不应当在出错时重新尝试.

```
int end_that_request_first(struct request *req, int success, int count);
void end_that_request_last(struct request *req);
```

使用 end_that_request_firest 来指示一个块 I/O 请求的一部分完成. 当那个函数返回 0，请求完成并且应当被传递给 end_that_request_last.

```
rq_for_each_bio(bio, request)
```

另一个用宏定义来实现的控制结构；它步入构成一个请求的每个 bio.

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct
scatterlist *list);
```

为一次 DMA 传送填充给定的散布表，用需要来映射给定请求中的缓冲的信息

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

make_request 函数的原型.

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

指示一个给定 bio 的完成. 这个函数应当只用在你的驱动直接获取 bio ，通过 make_request 函数从块层.

```
request_queue_t *blk_alloc_queue(int flags);
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

使用 blk_alloc_queue 来分配由定制的 make_request 函数使用的请求队列，. 那个函数应当使用 blk_queue_make_request 来设置.

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

一个命令准备函数的原型和设置函数，它可用来准备必要的硬件命令，在请求被传递给你的请求函数之前.

```
int blk_queue_init_tags(request_queue_t *queue, int depth, struct blk_queue_tag *tags);
```

```
int blk_queue_resize_tags(request_queue_t *queue, int new_depth);
int blk_queue_start_tag(request_queue_t *queue, struct request *req);
void blk_queue_end_tag(request_queue_t *queue, struct request *req);
struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);
void blk_queue_invalidate_tags(request_queue_t *queue);
```

驱动使用被标记的命令队列的支持函数.

第 17 章 网络驱动

我们已经讨论了字符和块驱动，现在准备好转移到网络世界里. 网络接口是第 3 类标准的 Linux 设备，本章描述它们如何与内核其他部分交互.

一个网络接口的在系统内的角色与一个被加载的块设备的角色类似. 一个块设备注册它的磁盘和工作方法到内核，随之通过它的请求函数按需求"发送"和"接收"块. 类似的，一个网络接口必须注册它自己到特定的内核数据结构中，以便在与外部世界交换报文时被调用.

在被加载的磁盘和报文递送接口之间有几个重要的区别. 首先，磁盘作为一个特殊的文件存在于 /dev 目录下，然而一个网络接口没有这样的入口点. 正常的文件操作(read, write, 等等)对于网络接口没有意义，因此不可能适用 Unix 的"一切皆文件"的方法给它们. 从而，网络接口存在于它们自己的名子空间里，并且对外输出了一套不同的操作.

尽管你可能会反驳说，应用程序在使用 socket 时可以使用 read 和 write 系统调用，这些系统调用作用于一个软件对象上，而它与接口是明显不同的. 几百个 socket 可以在同一个物理接口上复用.

但是两者最重要的不同在于，块驱动的运行只是响应来自内核的请求，但是网络驱动从外边异步地接收报文. 因此，不同于一个块驱动被要求向内核发送一个缓存区，网络设备要求向内核推送进入的报文. 网络驱动使用的内核接口为这个不同的操作模式而设计.

网络驱动也不得不准备支持很多的管理任务，例如设置地址，修改发送参数，以及维护流量和错误统计. 网络驱动使用的 API 反映了这种需要，并且因此，能看出一些与我们之前看到的接口的不同.

Linux 内核的网络子系统被设计成是完全独立于协议的. 这适用于网络协议(互联网协议 [IP]，相对于 IPX，或者其他协议)和硬件协议(以太网，相对的令牌环，等等). 一个网络驱动和内核互相作用在同一时间正确处理一个网络报文；这允许对驱动巧妙地隐藏了协议的信息，以及对协议隐藏了物理发送.

本章描述了网络接口如何适用于 Linux 内核的其他部分，并以一个基于内存模块化网络接口的形式提供了例子，它称做(你猜一下) snull. 为简化讨论，这个接口使用以太网硬件协议和发送 IP 报文. 你从测验 snull 中获得的知识已能够应用到非 IP 的协议上，并且编写一个非以太网驱动只是有极小的与实际网络协议相关的区别.

本章不讨论 IP 编号方案，网络协议，以及其他通用的网络概念. 这样的话题不是(常常地)驱动编写者所关心的，并且不可能提供一个满意的网络技术的概述在不足几百页里面. 建议感兴趣的读者去参考其他的描述网络方面的书籍.

在进入网络设备之前，提及一个技术方面的注意问题. 网络世界使用术语 octet 来表示一个 8 个位的组，它通常是网络设备和协议能理解的最小单元. 术语 byte 在这个上下文中极少遇到. 为紧跟标准用法，我们将使用 octet，在谈论网络设备的时候.

术语" header "也值得一提. 一个 header 是一组字节(错了，是 octet)，要安排到一个报文里，当它穿过网络子系统的各层时. 当一个应用程序通过一个 TCP socket 发送了一个数据块，网络子系统拆开数据，填充到报文里，在报文开始安上一个 TCP header 来描述每个报文在流里面的位置. 下面的协议层接着在 TCP header 之前安上一个 IP header，用来路由这个报文到它的目的地. 如果这个报文在类似以太网的介质上移动，一个以太网 header，由硬件来解析的，加在在余下的前面. 网络驱动(常常)不需要让自己去理睬高层的 header，但是它们经常必须参与硬件级别的 header 的创建.

17.1. snull 是如何设计的

本节谈论产生 snull 网络接口的设计概念. 尽管这个信息可能看来是边缘的使用, 不理解它在你运行例子代码时可能会导致问题.

首先, 也是最重要的, 设计的决定是例子接口应该保持独立于真实的硬件, 就像本书使用的大部分例子. 这个限制导致了一些构成环回接口的东西. snull 不是一个环回接口; 但是, 它模拟了与真实的远端主机间的对话, 以便更好演示编写一个网络驱动的任务. Linux 环回驱动实际是非常简单的; 它可在 drivers/net/lookback.c 找到.

snull 的另一个特性是它只支持 IP 通讯. 这是接口的内部工作的结果 -- snull 不得不查看里面并且解析报文来正确模拟一对硬件接口. 实际的接口不依赖于被发送的协议, 并且 snull 的这种限制不影响本章展示的代码片断.

17.1.1. 分配 IP 号

snull 模块创建了两个接口. 这些接口与一个简单的环回不同, 因为无论你通过其中一个接口发送什么都环回到另外一个, 而不是它自己. 它看起来好像你有两个外部连接, 但实际上是你的计算机在回答它自己.

不幸的是, 这个效果不能仅仅通过 IP 号码分配来完成, 因为内核不会通过接口 A 发送一个报文给它自己的接口 B, 它会利用环回通道而不是通过 snull. 为了能建立一个通过 snull 接口的通讯, 源和目的地址在实际传送中需要修改. 换句话说, 通过其中一个接口发送的报文应该被另一个收到, 但是外出报文的接受者不应当被认做是本地主机. 同样适用于接收到的报文的源地址.

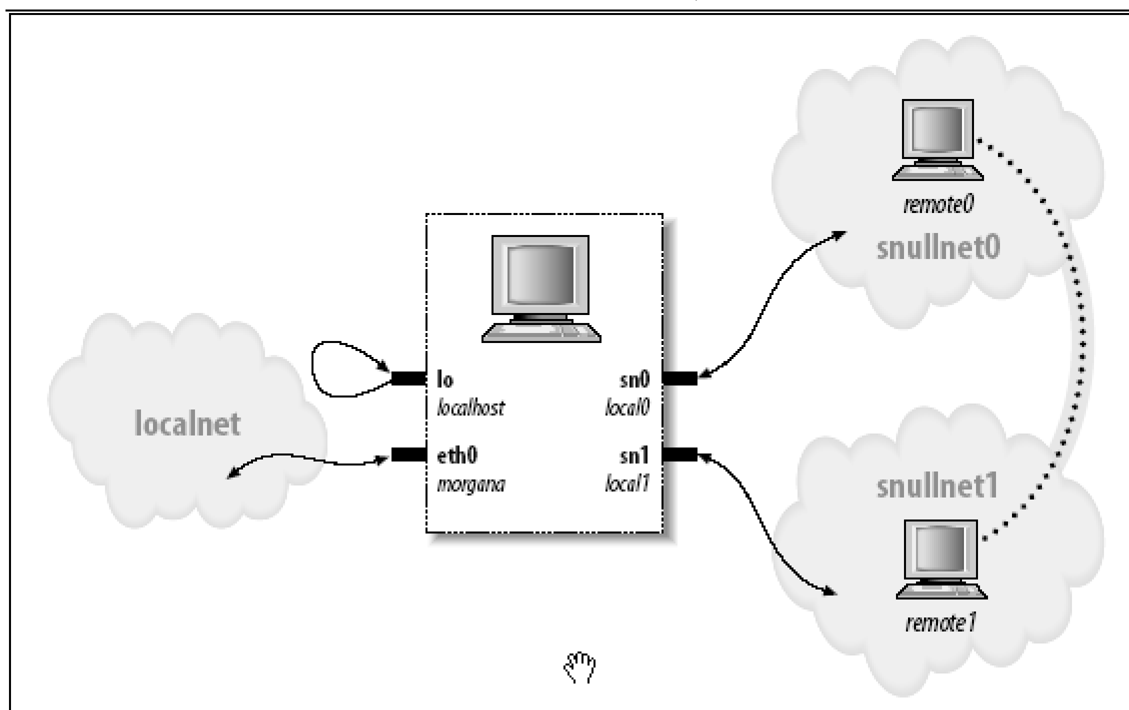
为获得这种"隐藏的环回", snull 接口翻转源地址和目的地址的第 3 个 octet 的最低有效位; 就是说, 它改变了 C 类 IP 编号的网络编号和主机编号. 网络方面的效果是发给网络 A(连接在 sn0 上, 第一个接口)的报文作为属于网络 B 的报文出现在 sn1 接口.

为避免处理太多编号, 我们分配符号名子给涉及到的 IP 编号:

- snullnet0 是连接到 sn0 接口的网络. 同样, snullnet1 是连接到 sn1. 这些网络的地址应当仅仅在第 3 个 octet 的最低有效位不同. 这些网络必须有 24 位的子网掩码.
- local0 是分配给 sn0 接口的 IP 地址; 它属于 snullnet0. 陪伴 sn1 的地址是 local1. local0 和 local1 必须在它们的第 3 octet 的最低有效位和第 4 octet 上不同.
- remote0 是在 snullnet0 的主机, 并且它的第 4 octet 与 local1 的相同. 任何发送给 remote0 的报文到达 local1, 在它的网络地址被接口代码改变之后. 主机 remote1 属于 snullnet1, 它的第 4 octet 与 local0 相同.

snull 接口的操作在图 主机如何看它的接口中描述, 其中每个接口的关联的主机名印在接口名的旁边.

图 17.1. 主机如何看它的接口



下面是网络编号的可能值. 一旦你把这些行放进 /etc/networks, 你可以使用名子来调用你的网络. 这些值选自保留做私人用途的编号范围.

```
snullnet0 192.168.0.0
snullnet1 192.168.1.0
```

下面的是一些可能的主机编号, 可放进 /etc/hosts 里面:

```
192.168.0.1   local0
192.168.0.2   remote0
192.168.1.2   local1
192.168.1.1   remote1
```

这些编号的重要特性是 local0 的主机部分与 remote1 的主机部分相同, local1 的主机部分和 remote0 的主机部分相同. 你可以使用完全不同的编号, 只要保持着这种关系.

但是要小心, 如果你的计算机以及连接到一个网络上. 你选择的编号可能是真实的互联网或者内联网的编号, 把它们安排给你的接口会阻止和这些真实的主机间的通讯. 例如, 尽管刚刚展示的这些编号不是可以路由的互联网编号, 它们也可能被你的私有网络已经在使用.

不管你选择什么编号, 你可以正确设置这些接口来操作, 通过发出下面的命令:

```
ifconfig sn0 local0
ifconfig sn1 local1
```

你可能需要添加网络掩码 255.255.255.0 参数, 如果选择的地址范围不是 C 类范围.

在此，接口的"远程"端点能够到达了. 下面的屏幕拷贝显示了一个主机如何到达 remote0 和 remote1 的，通过 snull 接口.

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

注意，你不能到达属于这两个网络的任何其他主机，因为报文被你的计算机丢弃了，在地址被修改和收到报文之后. 例如，一个发向 192.168.0.32 的报文将离开 sn0 并以 192.168.1.32 的目的地址出现在 sn1，这并不是这台主机的本地地址.

17.1.2. 报文的物理传送

只考虑数据传送的话，snull 接口属于以太网一类的.

snull 模拟以太网是因为大量的现存网络 -- 至少一个工作站所连接的网段 -- 是基于以太网技术的，它可能是 10base-T，100base-T，或者 千兆网. 另外，内核为以太网设备提供了一些通用的接口，没有理由不用它. 一个以太网设备的优势是如此的强以至于 plip 接口(使用打印机端口的接口)都声明自己是一个以太网设备.

snull 使用以太网设置的最后一个优势是你可以运行 tcpdump 在接口上来观察过往的报文. 使用 tcpdump 来观察接口是得知两个接口如何工作的有用途径.

如同我们之前提到的，snull 只处理 IP 报文. 这个限制来自这样的事实，snull 监听报文并且甚至修改它们，以便使代码工作. 代码修改了每个报文的源，目的和 IP header 的校验和，并不检查它是否实际承载着 IP 信息.

这种快而脏的数据修改毁坏了非 IP 报文. 如果你想通过 snull 递交其他协议，你必须修改模块的源代码.

17.2. 连接到内核

我们从分析 snull 的源码来查看网络驱动的结构开始. 把几个驱动的源码留在手边，对于下面的讨论和得知真实世界中的 Linux 网络驱动如何运行是会有帮助的.

17.2.1. 设备注册

当一个驱动模块加载进一个运行着的内核中，它请求资源并提供功能；这里没有新内容. 并且在资源是如何请求上也没有新东西. 驱动应当探测它的设备和它的硬件位置(I/O 端口和 IRQ 线) -- 但是不注册它们 --如在第 10 章的" 安装一个中断处理程序 "中所述. 一个网络驱动通过它的模块初始化函数注册的方式与字符和块驱动是不同的. 因为没有对等的主次编号给网络接口，一个网络驱动不请求这样一个号. 相反，驱动为每个刚刚探测到的接口在一个全局的网络设备列表里插入一个数据结构.

每个接口由一个结构 net_device 项来描述，它在 <linux/netdevice.h> 里定义. snull 驱动留有指向两个这样结构的指针，在一个简单数组里.

```
struct net_device *snull_devs[2];
```

net_device 结构，如同许多其他内核结构，包含一个 kobject，以及因此它可被引用计数并通过 sysfs 输出. 如同别的这样的结构，它必须动态分配. 进行这种分配的内核函数是 alloc_netdev，它有下列原型:

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                                void (*setup)(struct net_device *));
```

这里，sizeof_priv 是驱动的的"私有数据"区的大小；对于网络驱动，这个区是同 net_device 结构一起分配的. 实际上，这两个是是在一个大内存块中一起分配的，但是驱动作者应当假装不知道这一点. name 是这个接口的名子，如同用户空间看到的一样；这个名子可以有一个 printf 风格的 %d 在里面. 内核用下一个可用的接口号来替换这个 %d. 最后，setup 是一个初始化函数的指针，被调用来设置 net_device 结构的剩余部分. 我们即将进入这个初始化函数，但是现在，为强化起见，snull 以这样的方式分配它的两个设备结构:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",snull_init);
if (snull_devs[0] == NULL || snull_devs[1] == NULL)
    goto out;
```

象通常一样，我们必须检查返回值来确保分配成功.

网络子系统为各种接口提供了一些帮助函数，包裹着 alloc_netdev. 最通用的是 alloc_etherdev，定义在 <linux/etherdevice.h>:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

这个函数分配一个网络设备使用 eth%d 作为参数 name. 它提供了自己的初始化函数 (ether_setup)来设置几个 net_device 字段，使用对以太网设备合适的值. 因此，没有驱动提供的初始化函数给 alloc_etherdev；驱动应当只完成它要求的初始化，直接在一个成功的分配之后. 其他类型驱动的编写者可能想利用这些帮助函数的其中一个，例如 alloc_fcdev (定义在 <linux/fcdevice.h>) 为 fiber-channel 设备，alloc_fddidev (<linux/fddidevice.h>) 为 FDDI 设备，或者 aloc_trdev (<linux/trdevice.h>) 为令牌环设备.

snull 可以顺利使用 alloc_etherdev；我们选择使用 alloc_netdev 来代替，作为演示低层接口的方式，并且给我们控制安排给接口的名子.

一旦 net_device 结构完成初始化，完成这个过程就只是传递这个结构给 register_netdev. 在 snull 中，调用看来如同这样:

```
for (i = 0; i < 2; i++)
    if ((result = register_netdev(snull_devs[i])))
```

```
        printk("snull: error %i registering device \"%s\"\n",
               result, snull_devs[i]->name);
```

一些经常的注意问题这里提一下：在你调用 register_netdev 时，你的驱动可能会马上被调用来操作设备．因此，你不应当注册设备直到所有东西都已经完全初始化．

17.2.2. 初始化每一个设备

我们已经看到了 net_device 结构的分配和注册，但是我们越过了中间的完全初始化这个结构的步骤．注意 net_device 结构在运行时一直是放在一起；它不能如同一个 file_operations 或者 block_device_opreations 结构一样在编译时设置．必须在调用 register_netdev 之前完成初始化．net_device 结构又大又复杂；幸运的是，内核负责了一些以太网范围中的缺省值，通过 ether_setup 函数(由 alloc_etherdev 调用)．

因为 snull 使用 alloc_netdev，它有单独的初始化函数．该函数的核心(snull_init)如下：

```
ether_setup(dev); /* assign some of the fields */
dev->open = snull_open;
dev->stop = snull_release;
dev->set_config = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl = snull_ioctl;
dev->get_stats = snull_stats;
dev->rebuild_header = snull_rebuild_header;
dev->hard_header = snull_header;
dev->tx_timeout = snull_tx_timeout;
dev->watchdog_timeo = timeout;
/* keep the default flags, just add NOARP */
dev->flags |= IFF_NOARP;
dev->features |= NETIF_F_NO_CSUM;
dev->hard_header_cache = NULL; /* Disable caching */
```

上面的代码是对 net_device 结构的例行初始化；大部分是存储我们的各种驱动函数指针．代码的单个不寻常的特性是设置 IFF_NOARP 在 flags 里面．这个指出该接口不能使用 ARP．ARP 是一个低层以太网协议；它的工作是将 IP 地址转变成以太网介质存取控制 (MAC) 地址．因为由 snull 模拟的远程系统并不存在，就没人回答对它们的 ARP 请求．不想因为增加 ARP 实现使 snull 变复杂，我们选择标识接口作为不能处理这个协议．其中的对 hard_header_cache 赋值是同样理由：它关闭了这个接口的(不存在的) ARP 回答．这个主题在本章后面的" MAC 地址解析"一节中详述．

代码初始化也设置了几个和发送超时的处理有关的几个变量(tx_timeout 和 watchdog_time)．我们在"发送超时"一节完整地涉及这个主题．

我们现在看结构 net_device 的另一个成员，priv．它的角色近似于我们用在字符驱动上的 private_data 指针．不同于 fops->private_data，这个 priv 指针是随 net_device 结构一起分配的．也不鼓励直接存取 priv 成员，由于性能和灵活性的原因．当一个驱动需要存取私有数据指针，应当使用 netdev_priv 函数．因此，snull 驱动充满着这样的声明：

```
struct snull_priv *priv = netdev_priv(dev);
```

snull 模块声明了一个 snull_priv 数据结构来给 priv 使用:

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
    struct snull_packet *ppool;
    struct snull_packet *rx_queue; /* List of incoming packets */
    int rx_int_enabled;
    int tx_packetlen;
    u8 *tx_packetdata;
    struct sk_buff *skb;
    spinlock_t lock;

};
```

这个结构包括, 还有其他东西, 一个 net_device_stats 结构的实例, 这是放置接口统计量的标准地方. 下面的在 snull_init 中的各行分配并初始化 dev->priv:

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock);
snull_rx_ints(dev, 1); /* enable receive interrupts */
```

17.2.3. 模块卸载

模块卸载时没什么特别的. 模块的清理函数只是注销接口, 进行任何需要的内部清理, 释放 net_device 结构回系统.

```
void snull_cleanup(void)
{
    int i;

    for (i = 0; i < 2; i++) {
        if (snull_devs[i]) {
            unregister_netdev(snull_devs[i]);
            snull_teardown_pool(snull_devs[i]);
            free_netdev(snull_devs[i]);

        }
    }
    return;
}
```

对 unregister_netdev 的调用从系统中去除了接口; free_netdev 归还 net_device 结构给内核. 如果某个地方有对这个结构的引用, 它可能继续存在, 但是你的驱动不需要关心这个. 一旦你已经注销了接口, 内核不再调用它的方法.

注意我们的内部清理(在 snull_teardown_pool 里所做的)直到已经注销了设备后才能进行. 它必须, 但是, 在我们返回 net_device 结构给系统之前进行; 一旦我们已调用了 free_netdev, 我们再不能对这个设备或者我们的私有数据做任何引用.

17.3. net_device 结构的详情

net_device 结构处于网络驱动层的非常核心的位置并且值得完全的描述. 这个列表描述了所有成员, 更多的是提供了一个参考而不是用来备忘. 本章剩下的部分简要地描述了每个成员, 一旦它用在例子代码上, 因此你不需要不停地回看这一节.

17.3.1. 全局信息

结构 net_device 的第一部分是由下面成员组成:

char name[IFNAMSIZ];

> 设备名子. 如果名子由驱动设置, 包含一个 %d 格式串, register_netdev 用一个数替换它来形成一个唯一的名子; 分配的编号从 0 开始.

unsigned long state;

> 设备状态. 这个成员包括几个标志. 驱动正常情况下不直接操作这些标志; 相反, 提供了一套实用函数. 这些函数在我们进入驱动操作后马上讨论这些函数.

struct net_device *next;

> 全局列表中指向下一个设备的指针. 这个成员驱动不能动.

int (*init)(struct net_device *dev);

> 一个初始化函数. 如果设置了这个指针, 这个函数被 register_netdev 调用来完成对 net_device 结构的初始化. 大部分现代的网络驱动不再使用这个函数; 相反, 初始化在注册接口前进行.

17.3.2. 硬件信息

下面的成员包含了相对简单设备的低层硬件信息. 它们是早期 Linux 网络的延续; 大部分现代驱动确实使用它们(可能的例外是 if_port). 我们为完整起见在这里列出.

```
unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
```

> 设备内存信息. 这些成员持有设备使用的共享内存的开始和结束地址. 如果设备有不同的接收和发送内存, mem 成员由发送内存使用, rmem 成员由接收内存使用. rmem 成员在驱动之外从不被引用. 惯例上, 设置 end 成员, 所以 end - start 是可用的板上内存的数量.

unsigned long base_addr;

网络接口的 I/O 基地址. 这个成员, 如同前面的, 由驱动在设备探测时赋值. ifconfig 目录可用来显示或修改当前值. base_addr 可以当系统启动时在内核命令行中显式赋值(通过 netdev= 参数), 或者在模块加载时. 这个成员, 象上面描述过的内存成员, 内核不使用它们.

unsigned char irq;

安排的中断号. 当接口被列出时 ifconfig 打印出 dev->irq 的值. 这个值常常在启动或者加载时间设置并且在后来由 ifconfig 打印.

unsigned char if_port;

在多端口设备中使用的端口. 例如, 这个成员用在同时支持同轴线 (IF_PORT_10BASE2)和双绞线(IF_PORT_100BSAET)以太网连接. 完整的已知端口类型设置定义在 <linux/netdevie.h>.

unsigned char dma;

设备分配的 DMA 通道. 这个成员只在某些外设总线时有意义, 例如 ISA. 它不在设备驱动自身以外使用, 只是为了信息目的(在 ifconfig) 中.

17.3.3. 接口信息

有关接口的大部分信息由 ether_setup 函数正确设置(或者任何其他对给定硬件类型适合的设置函数). 以太网卡可以依赖这个通用的函数设置大部分这些成员, 但是 flags 和 dev_addr 成员是特定设备的, 必须在初始化时间明确指定.

一些非以太网接口可以使用类似 ether_setup 的帮助函数. deviers/net/net_init.c 输出了一些类似的函数, 包括下列:

void ltalk_setup(struct net_device *dev);

设置一个 LocalTalk 设备的成员

void fc_setup(struct net_device *dev);

初始化光通道设备的成员

void fddi_setup(struct net_device *dev);

配置一个光纤分布数据接口 (FDDI) 网络的接口

void hippi_setup(struct net_device *dev);

预备给一个高性能并行接口 (HIPPI) 的高速互连驱动的成员

```
void tr_setup(struct net_device *dev);
```

处理令牌环网络接口的设置

大部分设备会归于这些类别中的一类. 如果你的是全新和不同的, 但是, 你需要手工赋值下面的成员:

```
unsigned short hard_header_len;
```

硬件头部长度, 就是, 被发送报文前面在 IP 头之前的字节数, 或者别的协议信息. 对于以太网接口 hard_header_len 值是 14 (ETH_HLEN).

```
unsigned mtu;
```

最大传输单元 (MTU). 这个成员是网络层用作驱动报文传输. 以太网有一个 1500 字节的 MTU (ETH_DATA_LEN). 这个值可用 ifconfig 改变.

```
unsigned long tx_queue_len;
```

设备发送队列中可以排队的最大帧数. 这个值由 ether_setup 设置为 1000, 但是你可以改它. 例如, plip 使用 10 来避免浪费系统内存(相比真实以太网接口, plip 有一个低些的吞吐量).

```
unsigned short type;
```

接口的硬件类型. 这个 type 成员由 ARP 用来决定接口支持什么样的硬件地址. 对以太网接口正确的值是 ARPHRD_ETHER, 这是由 ether_setup 设置的值. 可认识的类型定义于 <linux/if_arp.h>.

```
unsigned char addr_len;
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char dev_addr[MAX_ADDR_LEN];
```

硬件 (MAC) 地址长度和设备硬件地址. 以太网地址长度是 6 个字节(我们指的是接口板的硬件 ID), 广播地址由 6 个 0xff 字节组成; ether_setup 安排成正确的值. 设备地址, 另外, 必须以特定于设备的方式从接口板读出, 驱动应当将它拷贝到 dev_addr. 硬件地址用来产生正确的以太网头, 在报文传递给驱动发送之前. snull 设备不使用物理接口, 它创造自己的硬件接口.

```
unsigned short flags;
int features;
```

接口标志(下面详述)

这个 flags 成员是一个位掩码, 包括下面的位值. IFF_ 前缀代表 "interface flags". 有些标志由内核管理, 有些由接口在初始化时设置来表明接口的能力和其他特性. 有效的标志, 对应于 <linux/if.h>, 有:

IFF_UP

对驱动这个标志是只读的. 内核打开它当接口激活并准备号传送报文时.

IFF_BROADCAST

这个标志(由网络代码维护)说明接口允许广播. 以太网板是这样.

IFF_DEBUG

这个标识了调试模式. 这个标志用来控制你的 printk 调用的复杂性或者用于其他调试目的. 尽管当前没有 in-tree 驱动使用这个标志, 它可以通过 ioctl 来设置和重置, 你的驱动可用它. misc-progs/netifdebug 程序可以用来打开或关闭这个标志.

IFF_LOOPBACK

这个标志应当只在环回接口中设置. 内核检查 IFF_LOOPBACK , 以代替硬连线 lo 名子作为一个特殊接口.

IFF_POINTOPOINT

这个标志说明接口连接到一个点对点链路. 它由驱动设置或者, 有时, 由 ifconfig. 例如, plip 和 PPP 驱动设置它.

IFF_NOARP

这个说明接口不能进行 ARP. 例如, 点对点接口不需要运行 ARP, 它只能增加额外的流量却没有任何有用的信息. snull 在没有 ARP 能力的情况下运行, 因此它设置这个标志.

IFF_PROMISC

这个标志设置(由网络代码)来激活混杂操作. 缺省地, 以太网接口使用硬件过滤器来保证它们只接收广播报文和直接到接口硬件地址的报文. 报文嗅探器, 例如 tcpdump, 在接口上设置混杂模式来存取在接口发送介质上经过的所有报文.

IFF_MULTICAST

驱动设置这个标志来表示接口能够组播发送. ether_setup 设置 IFF_MULTICAST 缺省地, 因此如果你的驱动不支持组播, 必须在初始化时清除这个标志.

IFF_ALLMULTI

这个标志告知接口接收所有的组播报文. 内核在主机进行组播路由时设置它, 前提是 IFF_MULTICAST 置位. IFF_ALLMULTI 对驱动是只读的. 组播标志在本章后面的"组播"一节中用到.

IFF_MASTER
IFF_SLAVE

这些标志由负载均衡代码使用. 接口驱动不需要知道它们.

IFF_PORTSEL
IFF_AUTOMEDIA

这些标志指出设备可以在多个介质类型间切换; 例如, 无屏蔽双绞线 (UTP) 和 同轴以太网电缆. 如果 IFF_AUTOMEDIA 设置了, 设备自动选择正确的介质. 特别地, 内核一个也不使用这 2 个标志.

IFF_DYNAMIC

这个标志, 由驱动设置, 指出接口的地址能够变化. 目前内核没有使用.

IFF_RUNNING

这个标志指出接口已启动并在运行. 它大部分是因为和 BSD 兼容; 内核很少用它. 大部分网络驱动不需要担心 IFF_RUNNING.

IFF_NOTRAILERS

在 Linux 中不用这个标志, 为了 BSD 兼容才存在.

当一个程序改变 IFF_UP, open 或者 stop 设备方法被调用. 进而, 当 IFF_UP 或者任何别的标志修改了, set_multicast_list 方法被调用. 如果驱动需要进行某些动作来响应标志的修改, 它必须在 set_multicast_list 中采取动作. 例如, 当 IFF_PROMISC 被置位或者复位, set_multicast_list 必须通知板上的硬件过滤器. 这个设备方法的责任在"组播"一节中讲解.

结构 net_device 的特性成员由驱动设置来告知内核关于任何的接口拥有的特别硬件能力. 我们将谈论一些这些特性; 别的就超出了本书范围. 完整的集合是:

NETIF_F_SG
NETIF_F_FRAGLIST

2 个标志控制发散/汇聚 I/O 的使用. 如果你的接口可以发送一个报文, 它由几个不同的内存段组成, 你应当设置 NETIF_F_SG. 当然, 你不得不实际实现发散/汇聚 I/O(我们在"发散/汇聚"一节中描述如何做). NETIF_F_FRAGLIST 表明你的接口能够处理分段的报文; 在 2.6 中只有环回驱动做这一点.

注意内核不对你的设备进行发散/汇聚 I/O 操作, 如果它没有同时提供某些校验和形式. 理由是, 如果内核不得不跨过一个分片的("非线性")的报文来计算校验和, 它可能也拷贝数据并同时接合报文.

NETIF_F_IP_CSUM
NETIF_F_NO_CSUM

NETIF_F_HW_CSUM

这些标志都是告知内核，不需要给一些或所有的通过这个接口离开系统的报文进行校验. 如果你的接口可以校验 IP 报文但是别的不行，就设置 NETIF_F_IP_CSUM. 如果这个接口不曾要求校验和，就设置 NETIF_F_NO_CSUM. 环回驱动设置了这个标志，snull 也设置；因为报文只通过系统内存传送，对它们来说没有机会(1 跳)被破坏，没有必要校验它们. 如果你的硬件自己做校验，设置 NETIF_F_HW_CWSUM.

NETIF_F_HIGHDMA

设置这个标志，如果你的设备能够对高端内存进行 DMA. 没有这个标志，所有提供给你的驱动的报文在低端内存分配.

NETIF_F_HW_VLAN_TX
NETIF_F_HW_VLAN_RX
NETIF_F_HW_VLAN_FILTER
NETIF_F_VLAN_CHALLENGED

这些选项描述你的硬件对 802.1q VLAN 报文的支持. VLAN 支持超出我们本章的内容. 如果 VLAN 报文使你的设备混乱(其实不应该)，设置标志 NETIF_F_VLAN_CHALLENGED.

NETIF_F_TSO

如果你的设备能够进行 TCP 分段卸载，设置这个标志. TSO 是一个我们在这不涉及的高级特性.

17.3.4. 设备方法

如同在字符和块驱动的一样，每个网络设备声明能操作它的函数. 本节列出能够对网络接口进行的操作. 有些操作可以留作 NULL，别的常常是不被触动的，因为 ether_setup 给它们安排了合适的方法.

网络接口的设备方法可分为 2 组：基本的和可选的. 基本方法包括那些必需的能够使用接口的；可选的方法实现更多高级的不是严格要求的功能. 下列是基本方法:

int (*open)(struct net_device *dev);

打开接口. 任何时候 ifconfig 激活它，接口被打开. open 方法应当注册它需要的任何系统资源(I/O 口，IRQ，DMA，等等)，打开硬件，进行任何别的你的设备要求的设置.

int (*stop)(struct net_device *dev);

停止接口. 接口停止当它被关闭. 这个函数应当恢复在打开时进行的操作.

int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);

起始报文的发送的方法. 完整的报文(协议头和所有)包含在一个 socket 缓存区 (sk_buff) 结构. socket 缓存在本章后面介绍.

int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);

用之前取到的源和目的硬件地址来建立硬件头的函数(在 hard_start_xmit 前调用). 它的工作是将作为参数传给它的信息组织成一个合适的特定于设备的硬件头. eth_header 是以太网类型接口的缺省函数, ether_setup 针对性地对这个成员赋值.

int (*rebuild_header)(struct sk_buff *skb);

用来在 ARP 解析完成后但是在报文发送前重建硬件头的函数. 以太网设备使用的缺省的函数使用 ARP 支持代码来填充报文缺失的信息.

void (*tx_timeout)(struct net_device *dev);

由网络代码在一个报文发送没有在一个合理的时间内完成时调用的方法, 可能是丢失一个中断或者接口被锁住. 它应当处理这个问题并恢复报文发送.

struct net_device_stats *(*get_stats)(struct net_device *dev);

任何时候当一个应用程序需要获取接口的统计信息, 调用这个方法. 例如, 当 ifconfig 或者 netstat -i 运行时. snull 的一个例子实现在"统计信息"一节中介绍.

int (*set_config)(struct net_device *dev, struct ifmap *map);

改变接口配置. 这个方法是配置驱动的入口点. 设备的 I/O 地址和中断号可以在运行时使用 set_config 来改变. 这种能力可由系统管理员在接口没有探测到时使用. 现代硬件正常的驱动一般不需要实现这个方法.

剩下的设备操作是可选的:

int weight;
int (*poll)(struct net_device *dev; int *quota);

由适应 NAPI 的驱动提供的方法, 用来在查询模式下操作接口, 中断关闭着. NAPI (以及 weight 成员) 在"接收中断缓解"一节中涉及.

void (*poll_controller)(struct net_device *dev);

在中断关闭的情况下, 要求驱动检查接口上的事件的函数. 它用于特殊的内核中的网络任务, 例如远程控制台和使用网络的内核调试.

int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);

处理特定于接口的 ioctl 命令. (这些命令的实现在"定制 ioclt 命令"一节中描述)相应的 net_device 结构中的成员可留为 NULL, 如果接口不需要任何特定于接口的命令.

void (*set_multicast_list)(struct net_device *dev);

当设备的组播列表改变和当标志改变时调用的方法. 详情见"组播"一节, 以及一个例子实现.

int (*set_mac_address)(struct net_device *dev, void *addr);

如果接口支持改变它的硬件地址的能力, 可以实现这个函数. 很多接口根本不支持这个能力. 其他的使用缺省的 eth_mac_adr 实现(在 deivers/net/net_init.c). eth_mac_addr 只拷贝新地址到 dev->dev_addr, 只在接口没有运行时作这件事. 使用 eth_mac_addr 的驱动应当在它们的 open 方法中自 dev->dev_addr 里设置硬件 MAC 地址.

int (*change_mtu)(struct net_device *dev, int new_mtu);

当接口的最大传输单元 (MTU) 改变时动作的函数. 如果用户改变 MTU 时驱动需要做一些特殊的事情, 它应当声明它的自己的函数; 否则, 缺省的会将事情做对. snull 有对这个函数的一个模板, 如果你有兴趣.

int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);

header_cache 被调用来填充 hh_cache 结构, 使用一个 ARP 请求的结果. 几乎全部类似以太网的驱动可以使用缺省的 eth_header_cache 实现.

int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);

在响应一个变化中, 更新 hh_cache 结构中的目的地址的方法. 以太网设备使用 eth_header_cache_update.

int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);

hard_header_parse 方法从包含在 skb 中的报文中抽取源地址, 拷贝到 haddr 的缓存区. 函数的返回值是地址的长度. 以太网设备通常使用 eth_header_parse.

17.3.5. 公用成员

结构 net_device 剩下的数据成员由接口使用来持有有用的状态信息. 有些是 ifconfig 和 netstat 用来提供给用户关于当前配置的信息. 因此, 接口应当给这些成员赋值:

unsigned long trans_start;
unsigned long last_rx;

保存一个 jiffy 值的成员. 驱动负责分别更新这些值, 当开始发送和收到一个报文时. trans_start 值被网络子系统用来探测发送器加锁. last_rx 目前没有用到, 但是驱动应当尽量维护这个成员以备将来使用.

int watchdog_timeo;

网络层认为一个传送超时发生前应当过去的最小时间(按 jiffy 计算), 调用驱动的 tx_timeout 函数.

void *priv;

filp->private_data 的对等者. 在现代的驱动里, 这个成员由 alloc_netdev 设置, 不应当直接存取; 使用 netdev_priv 代替.

struct dev_mc_list *mc_list;
int mc_count;

处理组播发送的成员. mc_count 是 mc_list 中的项数目. 更多细节见"组播"一节.

spinlock_t xmit_lock;
int xmit_lock_owner;

xmit_lock 用来避免对驱动的 hard_start_xmit 函数多个同时调用.
xmit_lock_owner 是已获得 xmit_lock 的 CPU 号. 驱动应当不改变这些成员的值.

结构 net_device 中有其他的成员, 但是网络驱动用不着它们.

17.4. 打开与关闭

我们的驱动可以在模块加载时或者内核启动时探测接口. 在接口能够承载报文前, 但是, 内核必须打开它并分配一个地址给它. 内核打开或者关闭一个接口对应 ifconfig 命令.

当 ifconfig 用来给接口安排一个地址, 它做 2 个任务. 第一, 它通过 ioctl(SIOCSIFADDR)(Socket I/O Control Set Interface Address) 来安排地址. 接着它设置 dev->flag 的 IFF_UP 位, 通过 ioctl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags) 来打开接口.

目前为止, ioctl(SIOCSIFADDR) 不做任何事. 没有驱动函数被调用 -- 这个任务是独立于设备的, 并且是内核实现它. 后面的命令 (ioctl(SIOCSIFFLAGS)), 但是, 为设备调用 open 方法.

相似地, 当接口关闭, ifconfig 使用 ioctl(SIOCSIFFLAGS) 来清除 IFF_UP, 并且 stop 方法被调用.

2 个设备方法都返回 0 在成功时, 并且出错时返回负值.

目前为止的实际代码，驱动不得不进行许多与字符和块驱动同样的任务. open 请求任何它需要的系统资源并且告知接口启动; stop 关闭接口并释放系统资源. 网络驱动必须进行一些附加的步骤在 open 时，但是.

第一，硬件 (MAC) 地址需要从硬件设备拷贝到 dev->dev_addr，在接口可以和外部世界通讯之前. 硬件地址接着在 open 时拷贝到设备. snull 软件接口在 open 里面安排它; 它只是使用了一个长为 ETH_ALEN 的字符串伪造了一个硬件号，ETH_ALEN 是以太网硬件地址长度.

open 方法应当也启动接口的发送队列(允许它接受发送报文)，一旦它准备好启动发送数据. 内核提供了一个函数来启动队列:

```
void netif_start_queue(struct net_device *dev);
```

snull 的 open 代码看来如下:

```
int snull_open(struct net_device *dev)
{
        /* request_region(), request_irq( ), ....  (like fops->open) */
        /*
        * Assign the hardware address of the board: use "\0SNULx", where
        * x is 0 or 1. The first byte is '\0' to avoid being a multicast
        * address (the first byte of multicast addrs is odd).
        */
        memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
        if (dev == snull_devs[1])
                dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
        netif_start_queue(dev);
        return 0;
}
```

如你所见，在缺乏真实硬件的情况下，在 *open* 方法中没什么可做. stop 方法也一样; 它只是反转 open 的操作. 因此，实现 stop 的函数常常称为 close 或者 release.

```
int snull_release(struct net_device *dev)
{
    /* release ports, irq and such -- like fops->close */
    netif_stop_queue(dev); /* can't transmit any more */
    return 0;
}
```

函数:

```
void netif_stop_queue(struct net_device *dev);
```

是 netif_start_queue 的对立面; 它标志设备为不能再发送任何报文. 这个函数必须在接口关闭(在 stop 方法中)时调用，但以可用于暂时停止发送，如下一节中解释的.

17.5. 报文传送

网络接口进行的最重要任务是数据发送和接收. 我们从发送开始, 因为它稍微易懂一些.

传送指的是通过一个网络连接发送一个报文的行为. 无论何时内核需要传送一个数据报文, 它调用驱动的 hard_start_stransmit 方法将数据放在外出队列上. 每个内核处理的报文都包含在一个 socket 缓存结构(结构 sk_buff)里, 定义见<linux/skbuff.h>. 这个结构从 Unix 抽象中得名, 用来代表一个网络连接, socket. 如果接口与 socket 没有关系, 每个网络报文属于一个网络高层中的 socket, 并且任何 socket 输入/输出缓存是结构 struct sk_buff 的列表. 同样的 sk_buff 结构用来存放网络数据历经所有 Linux 网络子系统, 但是对于接口来说, 一个 socket 缓存只是一个报文.

sk_buff 的指针通常称为 skb, 我们在例子代码和文本里遵循这个做法.

socket 缓存是一个复杂的结构, 内核提供了一些函数来操作它. 在"Socket 缓存"一节中描述这些函数; 现在, 对我们来说一个基本的关于 sk_buff 的事实就足够来编写一个能工作的驱动.

传给 hard_start_xmit 的 socket 缓存包含物理报文, 它应当出现在媒介上, 以传输层的头部结束. 接口不需要修改要传送的数据. skb->data 指向要传送的报文, skb->len 是以字节计的长度. 如果你的驱动能够处理发散/汇聚 I/O, 情形会稍稍复杂些; 我们在"发散/汇聚 I/O"一节中说它.

snull 报文传送代码如下; 网络传送机制隔离在另外一个函数里, 因为每个接口驱动必须根据特定的在驱动的硬件来实现它:

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data, shortpkt[ETH_ZLEN];
    struct snull_priv *priv = netdev_priv(dev);
    data = skb->data;
    len = skb->len;
    if (len < ETH_ZLEN) {
        memset(shortpkt, 0, ETH_ZLEN);
        memcpy(shortpkt, skb->data, skb->len);
        len = ETH_ZLEN;
        data = shortpkt;
    }
    dev->trans_start = jiffies; /* save the timestamp */
    priv->skb = skb; /* Remember the skb, so we can free it at interrupt time */

    /* actual deliver of data is device-specific, and not shown here */
    snull_hw_tx(data, len, dev);
    return 0; /* Our simple device can not fail */
}
```

传送函数, 因此, 只对报文进行一些合理性检查并通过硬件相关的函数传送数据. 注意, 但是, 要小心对待传送的报文比下面的媒介(对于 snull, 是我们虚拟的"以太网")支持的最小长度要短的情况. 许多 Linux 网络驱动(其他操作系统的也是)已被发现在这种情况下泄漏数据. 不是产生那种安全漏洞, 我们拷贝短报文到一个单独的数组, 这样我们可

以清楚地零填充到足够的媒介要求的长度. (我们可以安全地在堆栈中放数据, 因为最小长度 -- 60 字节 -- 是太小了).

hard_start_xmit 的返回值应当为 0 在成功时; 此时, 你的驱动已经负责起报文, 应当尽全力保证发送成功, 并且必须在最后释放 skb. 非 0 返回值指出报文这次不能发送; 内核将稍后重试. 这种情况下, 你的驱动应当停止队列直到已经解决导致失败的情况.

"硬件相关"的传送函数(snull_hw_tx)这里忽略了, 因为它完全是来实现了 snull 设备的戏法, 包括假造源和目的地址, 对于真正的网络驱动作者没有任何吸引力. 当然, 它呈现在例子源码里, 给那些想进入并看看它如何工作的人.

17.5.1. 控制发送并发

hard_start_xmit 函数由一个 net_device 结构中的自旋锁(xmit_lock)来保护避免并发调用. 但是, 函数一返回, 它有可能被再次调用. 当软件完成指导硬件报文发送的事情, 但是硬件传送可能还没有完成. 对 snull 这不是问题, 它使用 CPU 完成它所有的工作, 因此报文发送在传送函数返回前就完成了.

真实的硬件接口, 另一方面, 异步发送报文并且具备有限的内存来存放外出的报文. 当内存耗尽(对某些硬件, 会发生在一个单个要发送的外出报文上), 驱动需要告知网络系统不要再启动发送直到硬件准备好接收新的数据.

这个通知通过调用 netif_stop_queue 来实现, 这个前面介绍过的函数来停止队列. 一旦你的驱动已停止了它的队列, 它必须安排在以后某个时间重启队列, 当它又能够接受报文来发送了. 为此, 它应当调用:

```
void netif_wake_queue(struct net_device *dev);
```

这个函数如同 netif_start_queue, 除了它还刺探网络系统来使它又启动发送报文.

大部分现代的网络硬件维护一个内部的有多个发送报文的队列; 以这种方式, 它可以从网络上获得最好的性能. 这些设备的网络驱动必须支持在如何给定时间有多个未完成的发送, 但是设备内存能够填满不管硬件是否支持多个未完成发送. 任何时候当设备内存填充到没有空间给最大可能的报文时, 驱动应当停止队列直到有空间可用.

如果你必须禁止如何地方的报文传送, 除了你的 hard_start_xmit 函数(也许, 响应一个重新配置请求), 你想使用的函数是:

```
void netif_tx_disable(struct net_device *dev);
```

这个函数非常象 netif_stop_queue, 但是它还保证, 当它返回时, 你的 hard_start_xmit 方法没有在另一个 CPU 上运行. 队列能够用 netif_wake_queue 重启, 如常.

17.5.2. 传送超时

与真实硬件打交道的大部分驱动不得不预备处理硬件偶尔不能响应. 接口可能忘记它们在做什么, 或者系统可能丢失中断. 设计在个人机上运行的设备, 这种类型的问题是平常的.

许多驱动通过设置定时器来处理这个问题; 如果在定时器到期时操作还没结束, 有什么不对了. 网络系统, 本质上是一个复杂的由大量定时器控制的状态机的组合体. 因此, 网络代码是一个合适的位置来检测发送超时, 作为它正常操作的一部分.

因此, 网络驱动不需要担心自己去检测这样的问题. 相反, 它们只需要设置一个超时值, 在 net_device 结构的 watchdog_timeo 成员. 这个超时值, 以 jiffy 计, 应当足够长以容纳正常的发送延迟(例如网络媒介拥塞引起的冲突).

如果当前系统时间超过设备的 trans_start 时间至少 time-out 值, 网络层最终调用驱动的 tx_timeout 方法. 这个方法的工作是是进行清除问题需要的工作并且保证任何已经开始的发送正确地完成. 特别地, 驱动没有丢失追踪任何网络代码委托给它的 socket 缓存.

snull 有能力模仿发送器上锁, 由 2 个加载时参数控制的:

```
static int lockup = 0;
module_param(lockup, int, 0);

static int timeout = SNULL_TIMEOUT;
module_param(timeout, int, 0);
```

如果驱动使用参数 lockup=n 加载, 则模拟一个上锁, 一旦每 n 个报文传送了, 并且 watchdog_timeo 成员设为给定的时间值. 当模拟上锁时, snull 也调用 netif_stop_queue 来阻止其他的发送企图发生.

snull 发送超时处理看来如此:

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    PDEBUG("Transmit timeout at %ld, latency %ld\n", jiffies, jiffies - dev->trans_start);
    /* Simulate a transmission interrupt to get things moving */
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

当发生传送超时, 驱动必须在接口统计量中标记这个错误, 并安排设备被复位到一个干净的能发送新报文的状态. 当一个超时发生在 snull, 驱动调用 snull_interrupt 来填充"丢失"的中断并用 netif_wake_queue 重启队列.

17.5.3. 发散/汇聚 I/O

网络中创建一个发送报文的过程包括组合多个片. 报文数据必须从用户空间拷贝, 由网络协议栈各层使用的头部必须同时加上. 这个组合可能要求相当数量的数据拷贝. 但是, 如果注定要发送报文的网络接口能够进行发散/汇聚 I/O, 报文就不需要组装成一个单个块, 大量的拷贝可以避免. 发散/汇聚 I/O 也从用户空间启动"零拷贝"网络发送.

内核不传递发散的报文给你的 hard_start_xmit 方法除非 NETIF_F_SG 位已经设置到你的设备结构的特性成员中. 如果你已设置了这个标志, 你需要查看一个特殊的 skb 中的"shard info"成员来确定是否报文由一个单个片段或者多个组成, 并且如果需要就找出发散的片段. 一个特殊的宏定义来存取这个信息; 它是 skb_shinfo. 发送潜在的分片报文的第一步常常是看来如此的东东:

```
if (skb_shinfo(skb)->nr_frags == 0) {
    /* Just use skb->data and skb->len as usual */
}
```

nr_frags 成员告知多少片要用来建立这个报文. 如果它是 0, 报文存于一个单个片中, 可以如常使用 data 成员来存取. 但是, 如果它是非 0, 你的驱动必须历经并安排发送每一个单独的片. skb 结构的 data 成员方便地指向第一个片(在不分片情况下, 指向整个报文). 片的长度必须通过从 skb->len (仍然含有整个报文的长度) 中减去 skb->data_len 计算得来. 剩下的片会在称为 frags 的数组中找到, frags 在共享的信息结构中; frags 中每个入口是一个 skb_frag_struct 结构:

```
struct skb_frag_struct { struct page *page;
    __u16 page_offset;
    __u16 size;
};
```

如你所见, 我们又一次遇到 page 结构, 不是内核虚拟地址. 你的驱动应当遍历这些分片, 为 DMA 传送映射每一个, 并且不要忘记第一个分片, 它由 skb 直接指着. 你的硬件, 当然, 必须组装这些分片并作为一个单个报文发送它们. 注意, 如果你已经设置了 NETIF_F_HIGHDMA 特性标志, 一些或者全部分片可能位于高端内存.

17.6. 报文接收

从网络上接收报文比发送它要难一些, 因为必须分配一个 sk_buff 并从一个原子性上下文中递交给上层. 网络驱动可以实现 2 种报文接收的模式: 中断驱动和查询. 大部分驱动采用中断驱动技术, 这是我们首先要涉及的. 有些高带宽适配卡的驱动也可能采用查询技术; 我们在"接收中断缓解"一节中了解这个方法.

snull 的实现将"硬件"细节从设备独立的常规事务中分离. 因此, 函数 snull_rx 在硬件收到报文后从 snull 的"中断"处理中调用, 并且报文现在已经在计算机的内存中. snull_rx 收到一个数据指针和报文长度; 它唯一的责任是发走这个报文和运行附加信息给上层的网络代码. 这个代码独立于获得数据指针和长度的方式.

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
```

```
    /*
    *
    The packet has been retrieved from the transmission

    *
    medium. Build an skb around it, so upper layers can handle it
    */
    skb = dev_alloc_skb(pkt->datalen + 2);
    if (!skb) {

        if (printk_ratelimit())
            printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n"); priv-
>stats.rx_dropped++; goto out;
    }
    memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);

    /* Write metadata, and then pass to the receive level */
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
    priv->stats.rx_packets++;
    priv->stats.rx_bytes += pkt->datalen;
    netif_rx(skb);
out:
    return;
}
```

这个函数足够普通以作为任何网络驱动的一个模板，但是在你有信心重用这个代码段前需要一些解释.

第一步是分配一个缓存区来保存报文. 注意缓存分配函数 (dev_alloc_skb) 需要知道数据长度. 函数用这些信息来给缓存区分配空间. dev_alloc_skb 使用 atomic 优先级调用 kmalloc ， 因此它可以在中断时间安全使用. 内核提供了其他接口给 socket 缓存分配，但是它们不值得在此介绍; socket 缓存在"socket 缓存"一节中详细介绍.

当然，dev_alloc_skb 的返回值必须检查，snull 这样做了. 我们调用 printk_ratelimit 在抱怨失败之前，但是. 每秒钟产生成百上千的控制台消息是完全陷死系统和隐藏问题的真正源头的好方法; printk_ratelimit 帮助阻止这个问题，通过在有太多输出到了控制台时返回 0，事情需要慢下来一点.

一旦有一个有效的 skb 指针，通过调用 memcpy，报文数据被拷贝到缓存区; skb_put 函数更新缓存中的数据末尾指针并返回指向新建空间的指针.

如果你在编写一个高性能驱动，为一个可以进行完全总线占据 I/O 的接口，一个可能的优化值得在此考虑下. 一些驱动在报文接收前分配 sokcet 缓存，接着使接口将报文数据直接放入 socket 缓存空间. 网络层通过在可 DMA 的空间(如果你的设备设置了 NETIF_F_HIGHDMA 标志，这个空间有可能在高端内存)中分配所有 socket 缓存来配合这个策略. 这样避免了单独的填充 socket 缓存的拷贝操作，但是需要小心缓存区的大小，因为你无法提前知道进来的报文大小. change_mtu 方法的实现在这种情况下也重要，因为它允许驱动对最大报文大小改变作出响应.

网络层在搞懂报文的意思前需要清楚一些事情. 为此, dev 和 protocol 成员必须在缓存向上传递前赋值. 以太网支持代码输出一个帮助函数(eth_type_trans), 它发现一个合适值来赋给 protocol. 接着我们需要指出校验和要如何进行或者已经在报文上完成(snull 不需要做任何校验和). 对于 skb->ip_summed 可能的策略有:

CHECKSUM_HW

设备已经在硬件里做了校验. 一个硬件校验的例子使 APARC HME 接口.

CHECKSUM_NONE

校验和还没被验证, 必须由系统软件来完成这个任务. 这个是缺省的, 在新分配的缓存中.

CHECKSUM_UNNECESSARY

不要做任何校验. 这是 snull 和 环回接口的策略.

你可能奇怪为什么校验和状态必须在这里指定, 当我们已经在我们的 net_device 结构的特性成员中设置了标志. 答案是特性标志告诉内核我们的设备如何对待外出的报文. 它不用于进入的报文, 相反, 进入报文必须单独标记.

最后, 驱动更新它的统计计数来记录收到一个报文。 统计结构由几个成员组成; 最重要的是 rx_packet, rx_bytes, 和 tx_bytes, 分别含有收到的报文数目, 发送的数目, 和发送的字节总数. 所有的成员在"统计信息"一节中完全描述.

报文接收的最后一步由 netif_rx 进行, 它递交 socket 缓存给上层. 实际上 netif_rx 返回一个整数; NET_RX_SUCCESS(0) 意思是报文成功接收; 任何其他值指示错误. 有 3 个返回值 (NET_RX_CN_LOW, NET_RX_CN_MOD, 和 NET_RX_CN_HIGH)指出网络子系统的递增的拥塞级别; NET_RX_DROP 意思是报文被丢弃. 一个驱动在拥塞变高时可能使用这些值来停止输送报文给内核, 但是, 实际上, 大部分驱动忽略从 netif_rx 的返回值. 如果你在编写一个高带宽设备的驱动, 并且希望正确处理拥塞, 最好的办法是实现 NAPI, 我们在快速讨论中断处理后讨论它.

17.7. 中断处理

大部分硬件接口通过一个中断处理来控制. 硬件中断处理器来发出 2 种可能的信号: 一个新报文到了或者一个外出报文的发送完成了. 网络接口也能够产生中断来指示错误, 例如状态改变, 等等.

通常的中断过程能够告知新报文到达中断和发送完成通知的区别, 通过检查物理设备中的状态寄存器. snull 接口类似地工作, 但是它的状态字在软件中实现, 位于 dev->priv. 网络接口的中断处理看来如此:

```
static void snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int statusword;
    struct snull_priv *priv;
```

```
    struct snull_packet *pkt = NULL;
    /*
    *    As usual, check the "device" pointer to be sure it is
    *    really interrupting.
    *    Then assign "struct device *dev"
    */

    struct net_device *dev = (struct net_device *)dev_id;
    /* ... and check with hw if it's really ours */

    /* paranoid */
    if (!dev)
        return;

    /* Lock the device */
    priv = netdev_priv(dev);
    spin_lock(&priv->lock);

    /* retrieve statusword: real netdevices use I/O instructions */
    statusword = priv->status;
    priv->status = 0;
    if (statusword & SNULL_RX_INTR) {

        /* send it to snull_rx for handling */
        pkt = priv->rx_queue;
        if (pkt) {
            priv->rx_queue = pkt->next;
            snull_rx(dev, pkt);
        }
    }
    if (statusword & SNULL_TX_INTR) {
                /* a transmission is over: free the skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }

    /* Unlock the device and we are done */
    spin_unlock(&priv->lock);
    if (pkt) snull_release_buffer(pkt);     /* Do this outside the lock! */
    return;
}
```

中断处理的第一个任务是取一个指向正确 net_device 结构的指针. 这个指针通常来自作为参数收到的 dev_id 指针.

中断处理的有趣部分处理"发送结束"的情况. 在这个情况下, 统计量被更新, 调用 dev_kfree_skb 来返回 socket 缓存给系统. 实际上, 有这个函数的 3 个变体可以调用:

```
dev_kfree_skb(struct sk_buff *skb);
```

这个版本应当在你知道你的代码不会在中断上下文中运行时调用. 因为 snull 没有实际的硬件中断, 我们使用这个版本.

```
dev_kfree_skb_irq(struct sk_buff *skb);
```

如果你知道会在中断处理中释放缓存, 使用这个版本, 它对这个情况做了优化.

```
dev_kfree_skb_any(struct sk_buff *skb);
```

如果相关代码可能在中断或非中断上下文运行时, 使用这个版本.

最后, 如果你的驱动已暂时停止了发送队列, 这常常是用 netif_wake_queue 重启它的地方.

报文的接收, 相比于发送, 不需要特别的中断处理. 调用 snull_rx (我们已经见过)就是全部所需.

17.8. 接收中断缓解

当一个网络驱动如我们上面所述编写出来, 你的接口收到每个报文都中断处理器. 在许多情况下, 这是希望的操作模式, 它不是个问题. 然而, 高带宽接口能够在每秒内收到几千个报文. 这个样子的中断负载下, 系统的整体性能会受损害.

作为一个提高高端 Linux 系统性能的方法, 网络子系统开发者已创建了一种可选的基于查询的接口(称为 NAPI). [52]"查询"可能是一个不妥的字在驱动开发者看来, 他们常常看到查询是不灵巧和低效的. 查询是低效的, 但是, 仅仅在接口没有工作做的时候被查询. 当系统有一个处理大流量的高速接口时, 会一直有更多的报文来处理. 在这种情况下没有必要中断处理器; 时常从接口收集新报文是足够的.

停止接收中断能够减轻相当数量的处理器负载. 适应 NAPI 的驱动能够被告知不要输送报文给内核, 如果这些报文只是在网络代码里因拥塞而被丢弃, 这样能够在最需要的时候对性能有帮助. 由于各种理由, NAPI 驱动也比较少可能重排序报文.

不是所有的设备能够以 NAPI 模式操作, 但是. 一个 NAPI 适应的接口必须能够存储几个报文(要么在接口卡上, 要么在内存内 DMA 环). 接口应当能够禁止中断来接收报文, 却可以继续因成功发送或其他事件而中断. 有其他微妙的事情使得编写一个适应 NAPI 的驱动更有难度; 详情见内核源码中的 Documentation/networking/NAPI_HOWTO.txt.

相对少有驱动实现 NAPI 接口. 如果你在编写一个驱动给一个可能产生大量中断的接口, 但是, 花点时间来实现 NAPI 会被证明是很值得的.

snull 驱动, 当用非零的 use_napi 参数加载时, 在 NAPI 模式下操作. 在初始化时, 我们不得不建立一对格外的结构 net_device 的成员:

```
if (use_napi) {
    dev->poll   = snull_poll;
    dev->weight  = 2;
}
```

poll 成员必须设置为你的驱动的查询函数; 我们简短看一下 snull_poll. weight 成员描述接口的相对重要性: 有多少流量可以从接口收到, 当资源紧张时. 如何设置 weight 参数没有严格的规则; 依照惯例, 10 MBps 以太网接口设置 weight 为 16, 而快一些的接口使用 64. 你不能设置 weight 为一个超过你的接口能够存储的报文数目的值. 在 snull, 我们设置 weight 为 2, 作为一个演示不同报文接收的方法.

创建适应 NAPI 的驱动的下一步是改变中断处理. 当你的接口(它应当在接收中断使能下启动)示意有报文到达, 中断处理不应当处理这个报文. 相反, 它应当禁止后面的接收中断并告知内核到时候查询接口了. 在 snull 的"中断"处理里, 响应报文接收中断的代码已变为如下:

```
if (statusword & SNULL_RX_INTR) {
    snull_rx_ints(dev, 0); /* Disable further interrupts */
    netif_rx_schedule(dev);
}
```

当接口告诉我们有报文来了, 中断处理将其留在接口中; 此时需要的所有东西就是调用 netif_rx_schedule, 它使得我们的 poll 方法在后面某个时候被调用.

poll 方法有下面原型:

```
int (*poll)(struct net_device *dev, int *budget);
```

snull 的 poll 方法实现看来如此:

```
static int snull_poll(struct net_device *dev, int *budget)
{
    int npackets = 0, quota = min(dev->quota, *budget);
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
    struct snull_packet *pkt;

    while (npackets < quota && priv->rx_queue) {
        pkt = snull_dequeue_buf(dev);
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (! skb) {
            if (printk_ratelimit())
                printk(KERN_NOTICE "snull: packet dropped\n");
                 priv->stats.rx_dropped++;
                 snull_release_buffer(pkt);
                 continue;
        }
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY;       /* don't check it */
        netif_receive_skb(skb);

        /* Maintain stats */
        npackets++;
        priv->stats.rx_packets++;
```

```
        priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }

    /* If we processed all packets, we're done; tell the kernel and reenable ints */
    *budget -= npackets;
    dev->quota -= npackets;
    if (! priv->rx_queue) {
        netif_rx_complete(dev);
        snull_rx_ints(dev, 1);
        return 0;
    }
    /* We couldn't process everything. */
    return 1;

}
```

函数的中心部分是关于创建一个保持报文的 skb；这部分代码和我们之前在 snull_rx 中见到的一样. 但是，有些东西不一样：

- budget 参数提供了一个我们允许传给内核的最大报文数目. 在设备结构里，quota 成员给出了另一个最大值；poll 方法必须遵守这两个限制中的较小者. 它也应当以实际收到的报文数目递减 dev->quota 和 *budget. budget 值是当前 CPU 能够从所有接口收到的最多报文数目，而 quota 是一个每接口值，常常在初始化时安排给接口以 weight 为起始.
- 报文应当用 netif_receive_skb 递交内核，而不是 netif_rx.
- 如果 poll 方法能够在给定的限制内处理所有的报文，它应当重新使能接收中断，调用 netif_rx_complete 来关闭 查询，并且返回 0. 返回值 1 指示有剩下的报文需要处理.

网络子系统保证任何给定的设备的 poll 方法不会在多于一个处理器上被同时调用. 但是，poll 调用仍然可以与你的其他设备方法的调用并发.

---

[52] NAPI 代表"new API"；网络黑客们精于创建接口却疏于给它们起名.

17.9. 连接状态的改变

网络连接，根据定义，打交道本地系统之外的世界. 因此，它们常常受外界事件的影响，并且它们可能是短暂的东西. 网络子系统需要知道网络连接的上或下，它提供了几个驱动可用来传达这种信息的函数.

大部分涉及实际的物理连接的网络技术提供有一个载波状态；载波存在说明硬件存在并准备好. 以太网适配器，例如，在电线上感知载波信号；当一个用户绊倒一根电缆，载波消失，连接断开. 缺省地，网络设备假设有载波信号存在. 驱动可以明确改变这个状态，但是，使用这些函数：

```
void netif_carrier_off(struct net_device *dev);
```

```
void netif_carrier_on(struct net_device *dev);
```

如果你的驱动检测到它的一个设备载波丢失, 它应当调用 netif_carrier_off 来通知内核这个改变. 当载波回来时, 应当调用 netif_carrier_on. 一些驱动也调用 netif_carrier_off 当进行大的配置改变时(例如媒介类型); 一旦适配器已经完成复位它自身, 新载波被检测并且恢复流量.

一个整数函数也存在:

```
int netif_carrier_ok(struct net_device *dev);
```

它可用于测试当前载波状态(如同设备结构中所反映的);

17.10. Socket 缓存

我们现在已经涵盖到了大部分关于网络接口的问题. 还缺乏的是对 sk_buff 结构的详细描述.这个结构处于 Linux 内核网络子系统的核心, 我们现在介绍这个结构的重要成员和操作它们的函数.

尽管没有严格要求去理解 sk_buff 的内部, 能够查看它的内容的能力在你追踪问题和试图优化代码时是有帮助的. 例如, 如果你看 loopback.c, 你会发现一个基于对 sk_buff 内部了解的优化. 这里适用的通常的警告是: 如果你编写利用 sk_buff 结构的知识的代码, 你应当准备好在以后内核发行中它坏掉. 仍然, 有时性能优势值得额外的维护开销.

我们这里不会描述整个结构, 只是那些在驱动里可能用到的. 如果你想看到更多, 你可以查看 <linux/skbuff.h>, 那里定义了结构和函数原型. 关于如何使用这些成员和函数的额外的细节可以通过搜索内核源码很容易获取.

17.10.1. 重要成员变量

这里介绍的成员是驱动可能需要存取的. 以非特别的顺序列出它们.

```
struct net_device *dev;
```

接收或发送这个缓存的设备

```
union { /* ... */ } h;
union { /* ... */ } nh;
union { /*... */} mac;
```

指向报文中包含的各级的头的指针. union 中的某个成员都是一个不同数据结构类型的指针. h 含有传输层头部指针(例如, struct tcphdr *th); nh 包含网络层头部(例如 struct iphdr *iph); 以及 mac 包含链路层头部指针(例如 struct ethkr * ethernet).

如果你的驱动需要查看 TCP 报文的源和目的地址, 可以在 skb->h.th 中找到. 看头文件来找到全部的可以这样存取的头部类型.

注意网络驱动负责设置进入报文的 mac 指针. 这个任务正常是由 eth_type_trans 处理, 但是 非以太网驱动不得不直接设置 skb->mac.raw, 如同"非以太网头部"一节所示.

```
unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
```

用来寻址报文中数据的指针. head 指向分配内存的开始, data 是有效字节的开始(并且常常稍微比 head 大一些), tail 是有效字节的结尾, end 指向 tail 能够到达的最大地址. 查看它的另一个方法是可用缓存空间是 skb->end - skb->head, 当前使用的空间是 skb->tail - skb->data.

```
unsigned int len;
unsigned int data_len;
```

len 是报文中全部数据的长度, 而 data_len 是报文存储于单个片中的部分的长度. 除非使用发散/汇聚 I/O, data_len 成员的值为 0.

```
unsigned char ip_summed;
```

这个报文的校验和策略. 由驱动在进入报文上设置这个成员, 如在"报文接收"一节中描述的.

```
unsigned char pkt_type;
```

在递送中使用的报文分类. 驱动负责设置它为 PACKET_HOST (报文是给自己的), PACKET_OTHERHOST (不, 这个报文不是给我的), PACKET_BROADCAST, 或者 PACKET_MULTICAST. 以太网驱动不显式修改 pkt_type, 因为 eth_type_trans 为它们做.

```
shinfo(struct sk_buff *skb);
unsigned int shinfo(skb)->nr_frags;
skb_frag_t shinfo(skb)->frags;
```

由于性能的原因, 有些 skb 信息存储于一个分开的结构中, 它在内存中紧接着 skb. 这个"shared info"(这样命名是因为它可以在网络代码中多个 skb 拷贝中共享)必须通过 shinfo 宏定义来存取. 这个结构中有几个成员, 但是大部分超出本书的范围. 我们在"发散/汇聚 I/O"一节中见过 nr_frags 和 frags.

在结构中剩下的成员不是特别有趣. 它们用来维护缓存列表, 来统计 socket 拥有的缓存大小, 等等.

17.10.2. 作用于 socket 缓存的函数

使用一个 sk_buff 结构的网络驱动利用正式接口函数来操作它. 许多函数操作一个 socket 缓存; 这里是最有趣的几个:

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
```

分配一个缓存区. alloc_skb 函数分配一个缓存并且将 skb->data 和 skb->tail 都初始化成 skb->head. dev_alloc_skb 函数是使用 GFP_ATOMIC 优先级调用 alloc_skb 的快捷方法, 并且在 skb->head 和 skb->data 之间保留了一些空间. 这个数据空间用在网络层之间的优化, 驱动不要动它.

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

释放缓存. kfree_skb 调用由内核在内部使用. 一个驱动应当使用一种 dev_kfree_skb 的变体: 在非中断上下文中使用 dev_kfree_skb, 在中断上下文中使用 dev_kfree_skb_irq, 或者 dev_kfree_skb_any 在任何 2 种情况下.

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
```

更新 sk_buff 结构中的 tail 和 len 成员; 它们用来增加数据到缓存的结尾, 每个函数的返回值是 skb->tail 的前一个值(换句话说, 它指向刚刚创建的数据空间). 驱动可以使用返回值通过引用 memcpy(skb_put(...), data, len) 来拷贝数据或者一个等同的东东. 两个函数的区别在于 skb_put 检查以确认数据适合缓存, 而 __skb_put 省略这个检查.

```
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

递减 skb->data 和递增 skb->len 的函数. 它们与 skb_put 相似, 除了数据是添加到报文的开始而不是结尾. 返回值指向刚刚创建的数据空间. 这些函数用来在发送报文之前添加一个硬件头部. 又一次, __skb_push 不同在它不检查空间是否足够.

```
int skb_tailroom(struct sk_buff *skb);
```

返回可以在缓存中放置数据的可用空间数量. 如果驱动放了多于它能持有的数据到缓存中, 系统傻掉. 尽管你可能反对说一个 printk 会足够来标识出这个错误, 内存破坏对系统是非常有害的以至于开发者决定采取确定的动作. 实际中, 你不该需要检查可用空间, 如果缓存被正确地分配了. 因为驱动常常在分配缓存前获知报文的大小, 只有一个严重坏掉的驱动会在缓存中安放太多的数据, 这样出乱子就可当作一个应得的惩罚.

```
int skb_headroom(struct sk_buff *skb);
```

返回 data 前面的可用空间数量, 就是, 可以 "push" 给缓存多少字节.

```
void skb_reserve(struct sk_buff *skb, int len);
```

递增 data 和 tail. 这个函数可用来在填充数据前保留空间. 大部分以太网接口保留 2 个字节在报文的前面; 因此, IP 头对齐到 16 字节, 在 14 字节的以太网头后面. snull 也这样做, 尽管没有在"报文接收"一节中展现这个指令以避免在那时引入过多概念.

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

从报文的头部去除数据. 驱动不会需要使用这个函数, 但是为完整而包含在这儿. 它递减 skb->len 和递增 skb->data; 这是硬件头如何从进入报文开始被剥离.

```
int skb_is_nonlinear(struct sk_buff *skb);
```

返回一个真值, 如果这个 skb 分离为多个片为发散/汇聚 I/O.

```
int skb_headlen(struct sk_buff *skb);
```

返回 skb 的第一个片的长度(由 skb->data 指着).

```
void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
```

如果你必须从内核中的一个非线性 skb 直接存取片, 这些函数为你映射以及去映射它们. 使用一个原子性 kmap, 因此你不能一次映射多于一个片.

内核定义了几个其他的作用于 socket 缓存的函数, 但是它们是打算用于高层网络代码, 驱动不需要它们.

17.11. MAC 地址解析

以太网通讯的一个有趣的方面是如何将 MAC 地址(接口的唯一硬件 ID)和 IP 编号结合起来. 大部分协议有类似的问题, 但我们这里集中于类以太网的情况. 我们试图提供这个问题的完整描述, 因此我们展示三个情形: ARP, 无 ARP 的以太网头部(例如 plip), 以及非以太网头部.

17.11.1. 以太网使用 ARP

处理地址解析的通常方法是使用 Address Resolution Protocol (ARP). 幸运的是, ARP 由内核来管理, 并且一个以太网接口不需要做特别的事情来支持 ARP. 只要 dev->addr 和 dev->addr_len 在 open 时正确的赋值了, 驱动就不需要担心解决 IP 编号对应于 MAC 地址; ether_setup 安排正确的设备方法给 dev->hard_header 和 dev_rebuild_header.

尽管通常内核处理地址解析的细节(并且缓存结果), 它需要接口驱动来帮助建立报文. 毕竟, 驱动知道物理层头部细节, 然而网络代码的作者已经试图隔离内核其他部分. 为此,

内核调用驱动的 hard_header 方法使用 ARP 查询的结果来布置报文. 正常地, 以太网驱动编写者不需要知道这个过程 -- 公共的以太网代码负责了所有事情.

17.11.2. 不考虑 ARP

简单的点对点网络接口, 例如 plip, 可能从使用以太网头部中受益, 而避免来回发送 ARP 报文的开销. snull 中的例子代码也属于这一类的网络设备. snull 不能使用 ARP 因为驱动改变发送报文中的 IP 地址, ARP 报文也交换 IP 地址. 尽管我们可能轻易实现了一个简单 ARP 应答发生器, 更多的是演示性的来展示如何直接处理网络层头部.

如果你的设备想使用通常的硬件头而不运行 ARP, 你需要重写缺省的 dev->hard_header 方法. 这是 snull 的实现, 作为一个非常短的函数:

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
                 unsigned short type, void *daddr, void *saddr,
                 unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);
    eth->h_proto = htons(type);
    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest,  daddr ? daddr : dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1]  ^= 0x01;  /* dest is us xor 1 */
    return (dev->hard_header_len);
}
```

这个函数仅仅用内核提供的信息并把它格式成标准以太网头. 它也翻转目的以太网地址的 1 位, 理由下面叙述.

当接口收到一个报文, eth_type_trans 以几种方法来使用硬件头部. 我们已经在 snull_rx 看到这个调用.

```
skb->protocol = eth_type_trans(skb, dev);
```

这个函数抽取协议标识(ETH_P_IP, 在这个情况下)从以太网头; 它也赋值 skb->mac.raw, 从报文 data (使用 skb_pull)去掉硬件头部, 并且设置 skb->pkt_type. 最后一项在 skb 分配是缺省为 PACKET_HOST(指示报文是发向这个主机的), eth_type_trans 改变它来反映以太网目的地址: 如果这个地址不匹配接收它的接口地址, pkt_type 成员被设为 PACKET_OTHERHOST. 结果, 除非接口处于混杂模式或者内核打开了报文转发, netif_rx 丢弃任何类型为 PACKET_OTHERHOST 的报文. 因为这样, snull_header 小心地使目的硬件地址匹配接收接口.

如果你的接口是点对点连接, 你不会想收到不希望的多播报文. 为避免这个问题, 记住, 第一个字节的最低位(LSB)为 0 的目的地址是方向一个单个主机(即, 要么 PACKET_HOST, 要么 PACKET_OTHERHOST). plip 驱动使用 0xfc 作为它的硬件地址的第一个字节, 而 snull 使用 0x00. 两个地址都导致一个工作中的类似以太网的点对点连接.

17.11.3. 非以太网头部

我们刚刚看过硬件头部除目的地址外包含了一些信息，最重要的是通讯协议. 我们现在描述硬件头部如何用来封装相关的信息. 如果你需要知道细节，你可从内核源码里抽取它们或者从特定传送媒介的技术文档中. 大部分驱动编写者能够忽略这个讨论只是使用以太网实现.

值得一提的是不是所有信息都由每个协议提供. 一个点对点连接例如 plip 或者 snull 可能在不失去通用性的情况下避免传送这个以太网头部. hard_header 设备方法，由 snull_header 实现所展示的，接收自内核的递交的信息(协议级别和硬件地址). 它也在 type 参数中接收 16 位协议编号; IP，例如，标识为 ETH_P_IP. 驱动应该正确递交报文数据和协议编号给接收主机. 一个点对点连接可能它的硬件头部的地址，只传送协议编号，因为保证递交是独立于源和目的地址的. 一个只有 IP 的连接甚至可能不发送任何硬件头部.

当报文在连接的另一端被收到，接收函数应当正确设置成员 skb->protocol，skb->pkt_type，和 skb->mac.raw.

skb->mac.raw 是一个字符指针，由在高层的网络代码(例如，net/ipv4/arp.c)所实现的地址解析机制使用. 它必须指向一个匹配 dev->type 的机器地址. 设备类型的可能的值在 <linux/if_arp.h> 中定义; 以太网接口使用 ARPHRD_ETHER. 例如，这是 eth_type_trans 如何处理收到的报文的以太网头:

```
skb->mac.raw = skb->data;
skb_pull(skb, dev->hard_header_len);
```

在最简单的情况下(一个没有头的点对点连接)，skb->mac.raw 可指向一个静态缓存，包含接口的硬件地址，protocol 可设置为 ETH_P_IP，并且 packet_type 可让它是缺省的值 PACKET_HOST.

因为每个硬件类型是独特的，给出超出已经讨论的特别的设备是困难的. 内核中满是例子，但是. 例如，可查看 AppleTalk 驱动(drivers/net/appletalk/cops.c)，红外驱动(例如，driver/net/irds/smc_ircc.c)，或者 PPP 驱动(drivers/net/ppp_generic.c).

17.12. 定制 ioctl 命令

我们硬件看到给 socket 实现的 ioctl 系统调用; SIOCSIFADDR 和 SIOCSIFMAP 是 "socket ioctls" 的例子. 现在我们看看网络代码如何使用这个系统调用的 3 个参数.

当 ioctl 系统调用在一个 socket 上被调用，命令号是 <linux/sockios.h>中定义的符号中的一个，并且 sock_ioctl 函数直接调用一个协议特定的函数(这里"协议"指的是使用的主要网络协议，例如，IP 或者 AppleTalk).

任何协议层不识别的 ioctl 命令传递给设备层. 这些设备有关的 ioctl 命令从用户空间接收一个第 3 个参数，一个 struct ifreq*. 这个结构定义在 <linux/if.h>. SIOCSIFADDR 和 SIOCSIFMAP 命令实际上在 ifreq 结构上工作. SIOCSIFMAP 的额外参数，定义为 ifmap，只是 ifreq 的一个成员.

除了使用标准调用，每个接口可以定义它自己的 ioctl 命令. plip 接口，例如，允许接口通过 ioctl 修改它内部的超时值. socket 的 ioctl 实现认识 16 作为接口私有的个命令：SIOCDEVPRIVATE 到 SIOCDEVPRIVATE+15.[53]

当这些命令中的一个被识别，dev->do_ioctl 在相关的接口驱动中被调用. 这个函数接收与通用 ioctl 函数使用的相同的 struct ifreq * 指针.

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

ifr 指针指向一个内核空间地址，这个地址持有用户传递的结构的一个拷贝. 在 do_ioctl 返回之后，结构被拷贝回用户空间；因此，驱动可以使用这些私有命令接收和返回数据.

设备特定的命令可以选择使用结构 ifreq 中的成员，但是它们已经传达一个标准意义，并且不可能驱动使这个结构适应自己的需要. 成员 ifr_data 是一个 caddr_t 项(一个指针)，是打算用做设备特定的需要. 驱动和用来调用它的 ioctl 命令的程序应当一致地使用 ifr_data. 例如，ppp-stats 使用设备特定的命令来从 ppp 接口驱动获取信息.

这里不值得展示一个 do_ioctl 的实现，但是有了本章的信息和内核例子，你应当能够在你需要时编写一个. 注意，但是，plip 实现使用 ifr_data 不正确，不应当作为一个 ioctl 实现的例子.

---

[53] 注意，根据 <linux/sockios.h>，SIOCDEVPRIVATE 命令是被不赞成的. 应当使用什么来代替它们是不明确的，但是，并且不少在目录树中的驱动还使用它们.

17.13. 统计信息

驱动需要的最后一个方法是 get_stats. 这个方法返回一个指向给设备的统计的指针. 它的实现非常简单；展示过的这个即便在几个接口由同一个驱动管理时都好用，因为统计量驻留于设备数据结构内部.

```
struct net_device_stats *snull_stats(struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    return &priv->stats;
}
```

需要返回有意义统计的真正工作是分布在整个驱动中的，有各种成员量被更新. 下列列表展示了最有趣的结构 net_device_stats 中的成员：

```
unsigned long rx_packets;
unsigned long tx_packets;
```

接口成功传送的进入和出去报文的总和.

```
unsigned long rx_bytes;
```

unsigned long tx_bytes;

接口接收和发送的字节数.

unsigned long rx_errors;
unsigned long tx_errors;

接收和发送的错误数. 报文发送可能出错的事情是没有结束的, net_device_stats 结构包括 6 个计数器给特定的接收错误以及有 5 个给发送错误. 完整列表看 <<linux/netdevice.h>. 如果可能, 你的驱动调用维护详细的错误统计, 因为它们是对系统管理员试图追踪问题的最大帮助.

unsigned long rx_dropped;
unsigned long tx_dropped;

在接收和发送中丢失的报文数目. 当没有可用内存给报文数据时丢弃报文. tx_dropped 极少使用.

unsigned long collisions;

由于介质拥塞引起的冲突数目.

unsigned long multicast;

收到的多播报文数目.

值得重复一下, get_stats 方法可以在任何时候调用 -- 即便在接口关闭时 -- 因此只要 net_device 结构存在驱动必须保持统计信息.

17.14. 多播

一个多播报文是一个会被多个主机接收的网络报文, 但不是所有主机. 这个功能通过给一组主机分配特殊的硬件地址来获得. 发向一个特殊地址的报文应当被那个组当中的所有主机接收. 在以太网的情况下, 一个多播地址在目的地址的第一个字节的最低位为 1, 而每个设备板在它自己的硬件地址的这一位上为 0.

处理主机组和硬件地址的技巧由应用程序和内核处理, 接口驱动不必处理这个问题.

多播报文的传送是一个简单问题, 因为它们看起来就如同其他的报文. 接口发送它们通过通讯媒介, 不查看目的地址. 内核必须要安排一个正确的硬件目的地址; hard_header 设备方法, 如果定义了, 不必查看它安排的数据.

内核来跟踪在任何给定时间对哪些多播地址感兴趣. 这个列表可能经常改变, 因为它是在任何给定时间和按照用户意愿运行的应用程序的功能. 驱动的工作是接收感兴趣的多播地址列表并递交给内核任何发向这些地址的报文. 驱动如何实现多播列表是依赖于底层硬件是如何工作的. 典型地, 在多播的角度上, 硬件属于 3 类中的 1 种:

- 不能处理多播的接口. 这样的接口要么接收特别地发向它们的硬件地址(加上广播报文)的报文, 要么接收每一个报文. 它们只能通过接收每一个报文来接收多播报文, 因此, 潜在地压垮操作系统, 使用大量的"不感兴趣"报文. 你不经常认为这样的接口是有多播能力的, 驱动不会在 dev->flags 设置 IFF_MULTICAST.

  点对点接口是特殊情况, 因为它们一直接收每个报文, 不进行任何硬件过滤.

- 能够区别多播报文和其他报文(主机到主机, 或者广播). 这些接口能够被命令来接收每个多播报文, 让软件决定地址是否是主机感兴趣的. 这种情况下的开销是可接受的, 因为在一个典型网络上的多播报文的数目是少的.
- 可以进行硬件检测多播地址的接口. 可以传递一个多播地址的列表给这些接口, 这些地址的报文接收, 并忽略其他多播地址的报文. 对内核这是优化的情况, 因为它不浪费处理器时间来丢弃接口收到的"不感兴趣"的报文.

内核尽力利用高级接口的能力, 通过支持第 3 种设备类型, 它是最通用的. 因此, 内核通知驱动, 在任何有效多播地址列表发生改变时, 并且它传递新的列表给驱动, 因此它能够根据新的信息来更新硬件过滤器.

17.14.1. 多播的内核支持

对多播报文的支持有几项组成:一个设备方法, 一个数据结构, 以及设备标识:

void (*dev->set_multicast_list)(struct net_device *dev);

  设备方法, 在与设备相关的机器地址改变时调用. 它也在 dev->flags 被修改时调用, 因为一些标志(例如, IFF_PROMISC) 可能也要求你重新编程硬件过滤器. 这个方法接收一个 struct net_device 指针作为一个参数, 并返回 void. 一个对实现这个方法不感兴趣的驱动可以听任它为 NULL.

struct dev_mc_list *dev->mc_list;

  所有设备相关的多播地址的列表. 这个结构的实际定义在本节的末尾介绍.

int dev->mc_count;

  链表里的项数. 这个信息有些重复, 但是用 0 来检查 mc_count 是检查这个列表的有用的方法.

IFF_MULTICAST

  除非驱动在 dev->flags 中设置这个标志, 接口不会被要求来处理多播报文. 然而, 内核调用驱动的 set_multicast_list 方法, 当 dev->flags 改变时, 因为多播列表可能在接口未激活时改变了.

IFF_ALLMULTI

在 dev->flags 中设置的标志，网络软件来告知驱动从网络上接收所有多播报文．这发生在当多播路由激活时．如果标志设置了，dev->ma_list 不该用来过滤多播报文．

IFF_PROMISC

在 dev->flags 中设置的标志，当接口在混杂模式下．接口应当接收每个报文，不管 dev->ma_list．

驱动开发者需要的最后一点信息是 struct dev_mc_list 的定义，在 <linux/netdevice.h>:

```
struct dev_mc_list { struct dev_mc_list *next; /* Next address in list */
    __u8 dmi_addr[MAX_ADDR_LEN]; /* Hardware address */
    unsigned char  dmi_addrlen;  /* Address length */
    int  dmi_users;  /* Number of users */
    int  dmi_gusers;  /* Number of groups */
};
```

因为多播和硬件地址是独立于真正的报文发送，这个结构在网络实现中是可移植的，每个地址由一个字符串和一个长度标识，就像 dev->dev_addr．

17.14.2. 典型实现

描述 set_multicast_list 的设计的最好方法是给你看一些伪码．

下面的函数是一个典型函数实现在一个全特性(ff)驱动中．这个驱动是全模式的，它控制的接口有一个复杂的硬件报文过滤器，它能够持有一个主机要接收的多播地址表．表的最大尺寸是 FF_TABLE_SIZE．

所有以 ff_ 前缀的函数是给特定硬件操作的占位者:

```
void ff_set_multicast_list(struct net_device *dev) { struct dev_mc_list *mcptr;
    if (dev->flags & IFF_PROMISC) {
        ff_get_all_packets();
        return;
    }

    /* If there's more addresses than we handle, get all multicast packets and sort them
out in software. */
    if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {

        ff_get_all_multicast_packets();
        return;
    }

    /* No multicast? Just get our own stuff */
    if (dev->mc_count == 0) {
        ff_get_only_own_packets();
        return;
    }
```

```
    /* Store all of the multicast addresses in the hardware filter */
    ff_clear_mc_list();
    for (mc_ptr = dev->mc_list; mc_ptr; mc_ptr = mc_ptr->next)
        ff_store_mc_address(mc_ptr->dmi_addr);
    ff_get_packets_in_multicast_list();
}
```

这个实现可以简化，如果接口不能为进入报文存储多播表在硬件过滤器中. 这种情况下，FF_TABLE_SIZE 减为 0，并且代码的最后 4 行不需要了.

如同前面提过的，不能处理多播报文的接口不需要实现 set_multicast_list 方法来获取 dev->flags 改变的通知. 这个办法可能被称为一个"非特性的"(nf)实现. 实现非常简单，如下面代码所示:

```
void nf_set_multicast_list(struct net_device *dev)
{
    if (dev->flags & IFF_PROMISC)
        nf_get_all_packets();
    else
        nf_get_only_own_packets();
}
```

实现 IFF_PROMISC 是非常重要的，因为不这样用户就不能运行 tcpdump 或任何其他网络分析器. 如果接口运行一个点对点连接，另一方面，根本没有必要实现 set_multicast_list，因为用户接收每个报文.

17.15. 几个其他细节

本节涵盖了几个其他主题，对网络驱动作者感兴趣的. 在每种情况，我们试着简单指点你正确的方向. 获取了一个主题的完整描绘可能还需要花费一些时间深入内核源码.

17.15.1. 独立于媒介的接口支持

媒介独立接口(或 MII) 是一个 IEEE 802.3 标准，描述以太网收发器如何与网络控制器接口；很多市场上的产品遵守这个接口. 如果你在编写一个驱动为一个 MII 兼容控制器，内核输出了一个通用 MII 支持层，可能会使你易做一些.

为使用通用 MII 层，你应当包含 <linux/mii.h>. 你需要填充一个 mii_if_info 结构使用收发器的物理 ID 信息，如是否全双工有效. 还要求 mii_if_info 结构的 2 个方法:

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location);
void (*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

如你可能预料的，这些方法应当实现与你的特殊 MII 接口的通讯.

通用的 MII 代码提供一套函数，来查询和改变收发器的操作模式；许多设计成与 ethtool 工具一起工作(下一节描述). 在 <linux/mii.h> 和 drivers/net/mii.c 中查看细节.

17.15.2. ethtool 支持

ethtool 是一个实用工具, 设计来给系统管理员以大量的控制网络接口的操作. 用 ethtool, 可能来控制各种接口参数, 包括速度, 介质类型, 双工模式, DMA 环设置, 硬件校验和, LAN 唤醒操作, 等等, 但是只有当 ethtool 被驱动支持. ethtool 可以从 http://sf.net/projects/gkernel/. 下载.

对 ethtool 支持的相关声明可在 <linux/ethtool.h> 中找到. 它的核心是一个 ethtool_ops 类型的结构, 里面包含一个全部 24 个不同方法来支持 ethtool. 大部分这些方法是相对直接地; 细节看 <linux/ethtool.h>. 如果你的驱动使用 MII 层, 你能使用 mii_ethtool_gset 和 mii_ethtool_sset 来实现 get_settings 和 set_settings 方法, 分别地.

对于和你的设备一起工作的 ethtool, 你必须放置一个指向你的 ethtool_ops 结构的指针在 net_devcie 结构中. 宏定义 SET_ETHTOOL_OPS(在 <linux/netdevice.h> 中定义) 应当用作这个目的. 注意你的 ethtool 方法可能会在接口关闭时被调用.

17.15.3. netpoll

"netpoll" 是相对迟的增加到网络协议栈中; 它的目的是使内核能够发送和接收报文, 在完整的网络和 I/O 子系统不可用的情况下. 它用来给如远程网络控制台和远程内核调试等特色使用的. 无论如何, 你的驱动不必支持 netpoll, 但是它可能使你的驱动在某些情况下更有用. 在大部分情况下支持 netpoll 也相对容易.

实现 netpoll 的驱动应当实现 poll_controller 方法. 它的工作是跟上控制器上可能发生的任何东西, 在缺乏设备中断时. 几乎所有的 poll_controller 方法采用下面形式:

```
void my_poll_controller(struct net_device *dev)
{
    disable_device_interrupts(dev);
    call_interrupt_handler(dev->irq, dev, NULL);
    reenable_device_interrupts(dev);
}
```

poll_controller 方法, 实际上, 是简单模拟自给定设备的中断.

17.16. 快速参考

本节提供了本章中介绍的概念的参考. 也解释了每个驱动需要包含的头文件的角色. 在 net_device 和 sk_buff 结构中成员的列表, 但是, 这里没有重复.

#include <linux/netdevice.h>

定义 struct net_device 和 struct net_device_stats 的头文件, 包含了几个其他网络驱动需要的头文件.

```
struct net_device *alloc_netdev(int sizeof_priv, char *name, void (*setup)(struct net_device *);
```

```
struct net_device *alloc_etherdev(int sizeof_priv);
void free_netdev(struct net_device *dev);
```

分配和释放 net_device 结构的函数

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

注册和注销一个网络设备.

```
void *netdev_priv(struct net_device *dev);
```

获取网络设备结构的驱动私有区域的指针的函数.

```
struct net_device_stats;
```

持有设备统计的结构.

```
netif_start_queue(struct net_device *dev);
netif_stop_queue(struct net_device *dev);
netif_wake_queue(struct net_device *dev);
```

控制传送给驱动来发送的报文的函数. 没有报文被传送, 直到 netif_start_queue 被调用. netif_stop_queue 挂起发送, netif_wake_queue 重启队列并刺探网络层重启发送报文.

```
skb_shinfo(struct sk_buff *skb);
```

宏定义, 提供对报文缓存的"shared info"部分的存取.

```
void netif_rx(struct sk_buff *skb);
```

调用来通知内核一个报文已经收到并且封装到一个 socket 缓存中的函数.

```
void netif_rx_schedule(dev);
```

来告诉内核报文可用并且应当启动查询接口; 它只是被 NAPI 兼容的驱动使用.

```
int netif_receive_skb(struct sk_buff *skb);
void netif_rx_complete(struct net_device *dev);
```

应当只被 NAPI 兼容的驱动使用. netif_receive_skb 是对于 netif_rx 的 NAPI 对等函数; 它递交一个报文给内核. 当一个 NAPI 兼容的驱动已耗尽接收报文的供应, 它应当重开中断, 并且调用 netif_rx_complete 来停止查询.

```
#include <linux/if.h>
```

由 netdevice.h 包含, 这个文件声明接口标志(IFF_ 宏定义)和 struct ifmap, 它在网络驱动的 ioctl 实现中有重要地位.

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

前 2 个函数可用来告知内核是否接口上有载波信号. netif_carrier_ok 测试载波状态, 如同在设备结构中反映的.

```
#include <linux/if_ether.h>
ETH_ALEN
ETH_P_IP
struct ethhdr;
```

由 netdevice.h 包含, if_ether.h 定义所有的 ETH_ 宏定义, 用来代表字节长度(例如地址长度)以及网络协议(例如 IP). 它也定义 ethhdr 结构.

```
#include <linux/skbuff.h>
```

struct sk_buff 和相关结构的定义, 以及几个操作缓存的内联函数. 这个头文件由 netdevice.h 包含.

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

处理 socket 缓存的分配和释放的函数. 通常驱动应当使用 dev_ 变体, 其意图就是此目的.

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

添加数据到一个 skb 的函数; skb_put 在 skb 的尾部放置数据, 而 skb_push 放在开始. 正常版本进行检查以确保有足够的空间; 双下划线版本不进行检查.

```
int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);
```

进行 skb 中的空间管理的函数. skb_headroom 和 skb_tailroom 说明在开始和结尾分别有多少空间可用. skb_reserve 可用来保留空间, 在一个必须为空的 skb 开始.

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

skb_pull "去除" 数据从一个 skb, 通过调整内部指针.

```
int skb_is_nonlinear(struct sk_buff *skb);
```

如果这个 skb 是为发散/汇聚 I/O 分隔为几个片，函数返回一个真值.

```
int skb_headlen(struct sk_buff *skb);
```

返回 skb 的第一个片的长度，由 skb->data 指向.

```
void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
```

提供对非线性 skb 中的片直接存取的函数.

```
#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);
```

为以太网驱动设置大部分方法为通用实现的函数. 它还设置 dev->flags 和安排下一个可用的 ethx 给 dev->name，如果名子的第一个字符是一个空格或者 NULL 字符.

```
unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);
```

当一个以太网接口收到一个报文，这个函数被调用来设置 skb->pkt_type. 返回值是一个协议号，通常存储于 skb->protocol.

```
#include <linux/sockios.h>
SIOCDEVPRIVATE
```

前 16 个 ioctl 命令，每个驱动可为它们自己的私有用途而实现. 所有的网络 ioctl 命令都在 sockios.h 中定义.

```
#include <linux/mii.h>
struct mii_if_info;
```

声明和一个结构，支持实现 MII 标准的设备的驱动.

```
#include <linux/ethtool.h>
struct ethtool_ops;
```

声明和结构，使得设备与 ethtool 工具一起工作.

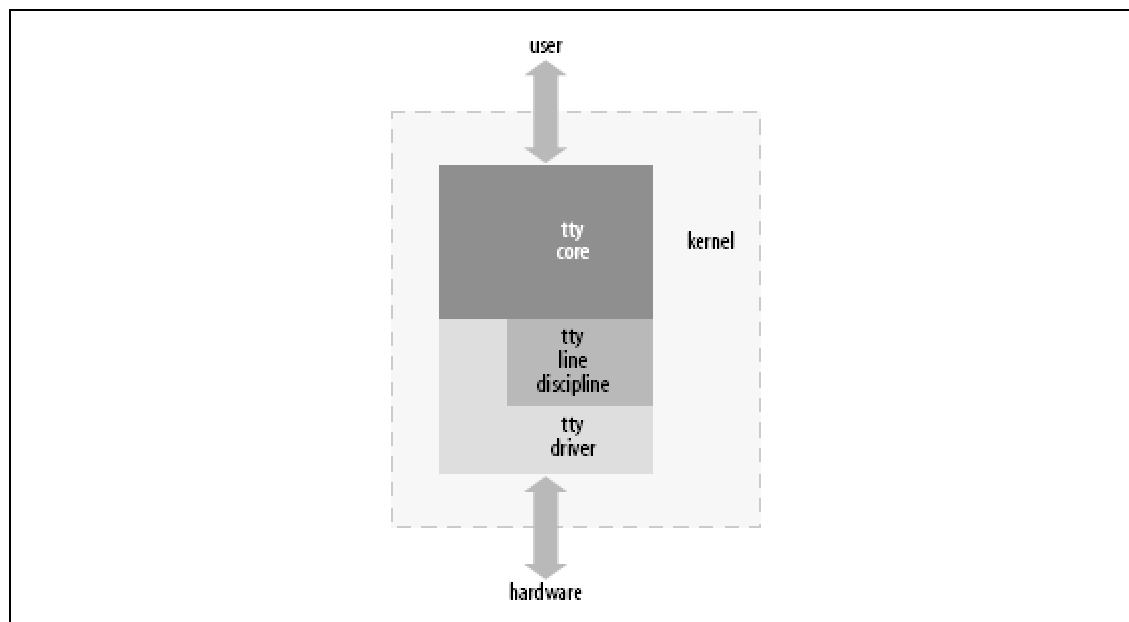
第 18 章 TTY 驱动

一个 tty 设备得名于电传打字机的很老的简称，并且起初只和连接到一台 UNIX 机器的物理或者虚拟终端有关联. 长时间以来，这个名子还逐渐表示任何串口类型的设备，因为终端连接也能够在这样的一个连接上建立. 一些物理 tty 设备的例子是串口，USB-串口 转换器，以及某些类型的需要特殊处理来正确工作的调制解调器(例如传统的 Win-Modem 类型设备). tty 虚拟设备支持虚拟控制台以用来登录到一台计算机，或者从键盘，或者从网络连接，或者通过一个 xterm 会话.

Linux tty 驱动的核心正好位于标准字符驱动级别之下，并且提供了一些特性集中在为使用终端类型设备提供一个接口. 这个核心负责控制跨越一个 tty 设备的数据流和数据格式. 这允许 tty 驱动以一种一致的方式集中于处理到硬件和出自硬件的数据，而不必担心如何控制对用户空间的接口. 为控制数据流，有几个不同的线路规程可以虚拟地"插入"任何一个 tty 设备. 这由不同的 tty 线路规程驱动来完成.

如同图 tty 核心概览 所示，tty 核心从一个用户获取将要发送给一个 tty 设备的数据. 它接着传递它到一个 tty 线路规程驱动，接着传递它到一个 tty 驱动. 这个 tty 驱动转换数据为可以发送给硬件的格式. 从 tty 硬件收到的数据向上回流通过 tty 驱动，进入 tty 线路规程驱动，再进入 tty 核心，在这里它被一个用户获取. 有时 tty 驱动直接和 tty 核心通讯，并且 tty 核心直接发送数据到 tty 驱动，但是常常 tty 线路规程有机会修改在 2 者之间发送的数据.

图 18.1. tty 核心概览



tty 驱动从未看见 tty 线路规程. 这个驱动不能直接和线路规程通讯，它甚至也不知道它存在. 驱动的工作是以硬件能够理解的方式格式化发送给它的数据，并且从硬件接收数据. tty 线路规程的工作是以特殊的方式格式化从一个用户或者硬件收到的数据. 这种格式化常常采用一个协议转换的形式，例如 PPP 和 Bluetooth.

有 3 种不同类型 tty 驱动: 控制台，串口，和 pty. 控制台和 pty 驱动硬件已经被编写以及可能是唯一需要的 tty 驱动的类型. 这使得任何使用 tty 核心来与用户和系统交互的新驱动作为串口驱动.

为知道什么类型的 tty 驱动当前被加载到内核以及什么 tty 设备当前存在, 查看 /proc/tty/drivers 文件. 这个文件包括一个当前存在的不同 tty 驱动的列表, 显示驱动的名子, 缺省的节点名子, 驱动的主编号, 这个驱动使用的次编号范围, 以及 tty 驱动的类型. 下面是一个这个文件的例子:

```
/dev/tty       /dev/tty      5     0     system:/dev/tty
/dev/console   /dev/console  5     1     system:console
/dev/ptmx      /dev/ptmx     5     2     system
/dev/vc/0      /dev/vc/0     4     0     system:vtmaster
usbserial      /dev/ttyUSB   188   0-254 serial
serial         /dev/ttyS     4     64-67 serial
pty_slave      /dev/pts      136   0-255 pty:slave
pty_master     /dev/ptm      128   0-255 pty:master
pty_slave      /dev/ttyp     3     0-255 pty:slave
pty_master     /dev/pty      2     0-255 pty:master
unknown        /dev/tty      4     1-63  console
```

/proc/tty/driver/ 目录给一些 tty 驱动包含了单独的文件, 如果它们实现这个功能. 缺省的串口驱动创建一个文件在这个目录中来展示许多串口特定的硬件信息. 如何在这个目录建立一个文件的信息后面描述.

所有的当前注册的以及在内核中出现的 tty 设备有它们自己的子目录在 /sys/class/tty 下面. 在那个子目录下, 有一个 "dev" 文件包含有分配给那个 tty 设备的主次编号. 如果这个驱动告知内核物理设备和关联到这个 tty 设备的驱动的所在, 它创建符号连接到它们. 这个树的一个例子是:

```
/sys/class/tty/
|-- console
| `-- dev
|-- ptmx
| `-- dev
|-- tty
| `-- dev
|-- tty0
| `-- dev
 ...
|-- ttyS1
| `-- dev
|-- ttyS2
| `-- dev
|-- ttyS3
| `-- dev
 ...
|-- ttyUSB0
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB0
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB1
| |-- dev
| |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB1
| `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
```

```
|-- ttyUSB2
| |-- dev
| |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB2
| `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
`-- ttyUSB3
|-- dev
|-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/ttyUSB3
`-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
```

18.1. 一个小 TTY 驱动

为解释 tty 核心如何工作，我们创建一个小 tty 驱动，可以被加载，以及写入读出，并且卸载. 任何一个 tty 驱动的主要数据结构是 struct tty_driver. 它用来注册和注销一个 tty 驱动到 tty 内核，在内核头文件 <linux/tty_driver.h> 中描述.

为创建一个 struct tty_driver，函数 alloc_tty_driver 必须用这个驱动作为参数而支持的 tty 设备号来调用. 这可使用下面的简短代码来完成:

```
/* allocate the tty driver */
tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);
if (!tiny_tty_driver)
 return -ENOMEM;
```

在 alloc_tty_driver 函数被成功调用后，struct tty_driver 应当用基于 tty 驱动的需要的正确信息被初始化. 这个结构包含很多不同成员，但不是为了有一个可工作的 tty 驱动而全部都必须被初始化. 这里有一个例子展示如何初始化这个结构并且建立足够的成员来创建一个工作的 tty 驱动. 它使用 tty_set_operations 函数来帮助拷贝驱动中定义的函数操作集合:

```
static struct tty_operations serial_ops = {
 .open = tiny_open,
 .close = tiny_close,
 .write = tiny_write,
 .write_room = tiny_write_room,
 .set_termios = tiny_set_termios,
};
...
 /* initialize the tty driver */
 tiny_tty_driver->owner = THIS_MODULE;
 tiny_tty_driver->driver_name = "tiny_tty";
 tiny_tty_driver->name = "ttty";
 tiny_tty_driver->devfs_name = "tts/ttty%d";
 tiny_tty_driver->major = TINY_TTY_MAJOR,
 tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,
 tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,
 tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,
 tiny_tty_driver->init_termios = tty_std_termios;
 tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
 tty_set_operations(tiny_tty_driver, &serial_ops);
```

上面列出的变量和函数，以及这个结构如何使用，在本章的剩下部分讲解.

为注册这个驱动到 tty 核心，struct tty_driver 必须传递到 tty_register_driver 函数:

```
/* register the tty driver */
retval = tty_register_driver(tiny_tty_driver);
if (retval)
{
        printk(KERN_ERR "failed to register tiny tty driver");
        put_tty_driver(tiny_tty_driver);
        return retval;
}
```

当调用 tty_register_driver，内核创建了所有的不同 sysfs tty 文件为这个 tty 驱动可能有的整个范围的次设备. 如果你使用 devfs (本书不涉及) 并且除非指定 TTY_DRIVER_NO_DEVFS 标志，devfs 文件也被创建. 这个标志可被指定如果你只想为这个实际在系统中存在的设备调用 tty_register_device，因此用户一直有一个内核中有的最新的设备视图，这就是 devfs 用户期望的.

在注册它自己后，这个驱动通过 tty_register_device 注册它控制的设备. 这个函数有 3 个参数:

- 一个指针指向这个设备所属的 struct tty_driver.
- 设备的次编号
- 一个指针指向这个 tty 设备所绑定的 struct device. 如果这个 tty 设备没绑定到任何一个 struct device，这个参数可被设为 NULL.

我们的驱动一次注册所有的 tty 设备，因为它们是虚拟的并且没有绑定到任何一个物理设备:

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
        tty_register_device(tiny_tty_driver, i, NULL);
```

为从 tty 核心注销这个驱动，所有的通过调用 tty_register_device 而注册的 tty 设备需要使用对 tty_unregister_device 的调用来清理. 接着 struct tty_driver 必须使用一个 tty_unregister_driver 调用来注销.

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
        tty_unregister_device(tiny_tty_driver, i);
tty_unregister_driver(tiny_tty_driver);
```

18.1.1. 结构 struct termios

在 struct tty_driver 中的 init_termios 变量是一个 struct termios. 这个变量被用来提供一个健全的线路设置集合，如果这个端口在被用户初始化前使用. 驱动初始化这个变量使用一个标准的数值集，它拷贝自 tty_std_termios 变量. tty_std_termos 在 tty 核心被定义为:

```
struct termios tty_std_termios = {
 .c_iflag = ICRNL | IXON,
 .c_oflag = OPOST | ONLCR,
 .c_cflag = B38400 | CS8 | CREAD | HUPCL,
 .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
 ECHOCTL | ECHOKE | IEXTEN,
 .c_cc = INIT_C_CC
};
```

这个 struct termios 结构用来持有所有的当前线路设置，给这个 tty 设备的一个特定端口. 这些线路设置控制当前波特率，数据大小，数据流控设置，以及许多其他值. 这个结构的不同成员是:

tcflag_t c_iflag;

输入模式标志

tcflag_t c_oflag;

输出模式标志

tcflag_t c_cflag;

控制模式标志

tcflag_t c_lflag;

本地模式标志

cc_t c_line;

线路规程类型

cc_t c_cc[NCCS];

一个控制字符数组

所有的模式标志被定义为一个大的位段. 模式的不同值，以及它们用在哪里，可以见在任何 Linux 发布中都有的 termios 手册页. 内核提供了一套有用的宏定义来获得不同的位. 这些宏定义在头文件 include/linux/tty.h 中定义.

所有的在 tiny_tty_driver 变量中定义的成员有必要有一个工作的 tty 驱动. owner 成员是为了防止 tty 驱动在 tty 端口打开时被卸载. 在以前的内核版本，它由 tty 驱动自己负责处理模块引用计数逻辑. 但是内核程序员认为可能有困难来解决所有的不同的可能的竞争条件，因此 tty 核心为 tty 驱动处理所有的这样的控制..

driver_name 和 name 成员看起来非常相似，然而用于不同用途. driver_name 变量应当设为某个简单的，描述性的并且和内核中所有 tty 驱动中是唯一的值. 这是因为它在 /proc/tty/drivers 文件中出现来描述这个驱动给用户，以及在当前已加载的 tty 驱动的 sysfs tty 类目录. name 成员用来定义一个名子给单独的分配给这个 tty 驱动的

tty 节点在 /dev 树中. 这个字符串用来创建一个 tty 设备通过在这个字串的后面追加在使用的 tty 设备号. 它还用来创建一个设备名子在 sysfs /sys/class/tty 目录中. 如果 devfs 在内核中被使能, 这个名子应当包含任何这个 tty 驱动想被放入的子目录. 作为一个例子, 内核中的串口驱动设置这个 name 成员为 tts/ 如果 devfs 被使能, ttyS 如果它没有被使能. 这个字串也显示在 /proc/tty/drivers 文件中.

如同我们提及的, /proc/tty/drivers 文件展示所有的当前注册的 tty 驱动. 在内核中注册的 tiny_tty 驱动并且没有 devfs, 这个文件看来如下:

```
$ cat /proc/tty/drivers
tiny_tty       /dev/ttty      240   0-3     serial
usbserial      /dev/ttyUSB    188   0-254   serial
serial         /dev/ttyS      4     64-107  serial
pty_slave      /dev/pts       136   0-255   pty:slave
pty_master     /dev/ptm       128   0-255   pty:master
pty_slave      /dev/ttyp      3     0-255   pty:slave
pty_master     /dev/pty       2     0-255   pty:master
unknown        /dev/vc/       4     1-63    console
/dev/vc/0      /dev/vc/0      4     0       system:vtmaster
/dev/ptmx      /dev/ptmx      5     2       system
/dev/console   /dev/console   5     1       system:console
/dev/tty       /dev/tty       5     0       system:/dev/tty
```

还有, 当 tny_tty driver 被注册到 tty 核心, sysfs 目录 /sys/class/tty 看来有些象下面:

```
$ tree /sys/class/tty/ttty*
/sys/class/tty/ttty0
`-- dev
/sys/class/tty/ttty1
`-- dev
/sys/class/tty/ttty2
`-- dev
/sys/class/tty/ttty3
`-- dev

$ cat /sys/class/tty/ttty0/dev
240:0
```

major 变量描述这个驱动的主编号是什么. type 和 subtype 变量声明这个驱动是什么 tty 驱动. 对于我们的例子, 我们是一个"正常"类型的串口驱动. 一个 tty 驱动的唯一的其他子类型可能是一个 "callout" 类型. callout 设备传统上用来控制一个设备的线路设置. 数据应当通过一个设备节点被发送和接收, 并且任何路线设置改变应当被发送给一个不同的设备节点, 它是这个 callout 设备. 这要求使用 2 个次编号为每个 tty 设备. 感激地, 所有的驱动既处理数据也处理线路设置在同一个设备节点, 并且这个 callout 类型很少用在新驱动中.

tty 驱动和 tty 核心都使用 flags 变量来指示驱动的当前状态和它是什么类型 tty 驱动. 几个在测试或者操作 flags 时你必须使用的位掩码宏被定义了. flags 变量中的 3 个位可被驱动设置:

TTY_DRIVER_RESET_TERMIOS

这个标志说明 tty 核心复位了 termios 设置，无论何时最后一个进程已关闭这个设备．对于控制台和 pty 驱动这是有用的．例如，假定用户留置一个终端在一个奇怪的状态．在设置了这个标志时，这个终端被复位为一个正常值当用户注销或者控制个会话的进程被"杀掉"．

TTY_DRIVER_REAL_RAW

这个标志说明 tty 驱动保证发送奇偶或者坏字符通知给线路规程．这允许线路规程以一种更快的方式来处理接收到的字符，因为它不必查看从 tty 驱动收到的每个字符．因为速度的得益，这个值常常为所有 tty 驱动设置．

TTY_DRIVER_NO_DEVFS

这个标志说明当调用 tty_register_driver 时，tty 核心不创建任何 devfs 入口给这个 tty 设备．这对任何动态创建和销毁次设备的驱动都是有益的．设置这个的驱动的例子是这个 USB-到-串口 驱动，USB 猫驱动，USB 蓝牙 tty 驱动，以及好多标准串口设备．

当 tty 驱动后来想注册一个特殊的 tty 设备到 tty 核心，它必须调用 tty_register_device，有一个指针到这个 tty 驱动，并且设备的次编号已被创建．如果这个没有完成，tty 核心仍然传递所有的调用到这个 tty 驱动，但是一些内部的 tty 相关的功能可能不存在．这个包括新 tty 设备的 /sbin/hotplug 通知和 tty 设备的 sysfs 表示．当注册的 tty 设备从机器中被移出，tty 驱动必须调用 tty_unregister_device．

The one remaining bit in this variable is controlled by the tty core and is called TTY_DRIVER_INSTALLED. This flag is set by the tty core after the driver has been regis-tered and should never be set by a tty driver.
这个变量中剩下的一位被 tty 核心控制，被称为 TTY_DRIVER_INSTALLED．这个标志被 tty 核心在驱动已注册后设置并且应当从不被 tty 驱动设置．

18.2. tty_driver 函数指针

最终，tiny_tty 驱动声明了 4 个函数指针．

18.2.1. open 和 close

open 函数被 tty 核心调用，当一个用户对这个 tty 驱动被分配的设备节点调用 open 时．tty 核心使用一个指向分配给这个设备的 tty_struct 结构的指针调用它，还用一个文件指针．这个 open 成员必须被一个 tty 驱动为它能正确工作而设置；否则，-ENODEV 被返回给用户当调用 open 时．

当调用这个 open 函数，tty 驱动被期望或者保存一些传递给它的 tty_struct 变量中的数据，或者保存一个可以基于端口次编号来引用的静态数组中的数据．这是有必要的，所以 tty 驱动知道哪个设备在被引用当以后的 close，write，和其他函数被调用时．

tiny_tty 驱动保存一个指针在 tty 结构中，如同下面代码所见到：

```
static int tiny_open(struct tty_struct *tty, struct file *file)
{
        struct tiny_serial *tiny;
        struct timer_list *timer;
        int index;

        /* initialize the pointer in case something fails */
        tty->driver_data = NULL;

        /* get the serial object associated with this tty pointer */
        index = tty->index;
        tiny = tiny_table[index];
        if (tiny == NULL)
        {
                /* first time accessing this device, let's create it */
                tiny = kmalloc(sizeof(*tiny), GFP_KERNEL);
                if (!tiny)

                        return -ENOMEM;
                init_MUTEX(&tiny->sem);
                tiny->open_count = 0;
                tiny->timer = NULL;

                tiny_table[index] = tiny;
        }

        down(&tiny->sem);
        /* save our structure within the tty structure */
        tty->driver_data = tiny;
        tiny->tty = tty;
```

在这个代码中，tiny_serial 结构被保存在 tty 结构中. 这允许 tiny_write, tiny_write_room, 和 tiny_close 函数来获取 tiny_serial 结构和正确操作它.

tiny_serial 结构定义为：

```
struct tiny_serial
{
        struct tty_struct *tty;     /* pointer to the tty for this device */
        int open_count;             /* number of times this port has been opened */
        struct semaphore  sem;      /* locks this structure */
        struct timer_list  *timer;
};
```

如同我们已见到的，open_count 变量初始化为 0 在第一次打开端口的 open 调用中. 这是一个典型的引用计数，因为一个 tty 驱动的 open 和 close 函数可能对同一个设备多次调用以便多个进程来读写数据. 为正确处理所有的事情，必须保持一个这个端口被打开或者关闭的次数计数；这个驱动递增和递减这个计数在打开使用时. 当打开第一次被打开，

任何必要的硬件初始化和内存分配可以做. 当端口被最后一次关闭, 任何必要的硬件关闭和内存清理可以做.

tiny_open 函数的剩下部分展示了如何跟踪设备被打开的次数:

```
++tiny->open_count;
if (tiny->open_count == 1)
{
        /* this is the first time this port is opened */
        /* do any hardware initialization needed here */
}
```

open 函数必须返回或者一个负的错误号如果发生事情阻止了成功打开, 或者一个 0 来表示成功.

close 函数指针被 tty 核心调用, 在用户对前面使用 open 调用而创建的文件句柄调用 close 时. 这表示设备应当在这次被关闭. 但是, 因为 open 函数可被多次调用, close 函数也可多次调用. 因此这个函数应当跟踪它被调用的次数来决定是否硬件应当在此次真正被关闭. tiny_tty 驱动做这个使用下面的代码:

```
static void do_close(struct tiny_serial *tiny)
{
        down(&tiny->sem);

        if (!tiny->open_count)
        {
                /* port was never opened */
                goto exit;

        }
        --tiny->open_count;
        if (tiny->open_count <= 0)
        {
                /* The port is being closed by the last user. */
                /* Do any hardware specific stuff here */
                /* shut down our timer */
                del_timer(tiny->timer);
        }
exit:
        up(&tiny->sem);
}

static void tiny_close(struct tty_struct *tty, struct file *file)
{
        struct tiny_serial *tiny = tty->driver_data;

        if (tiny)
                do_close(tiny);
}
```

tiny_close 函数只是调用 do_close 函数来完成实际的关闭设备工作. 因此关闭逻辑不必在这里和驱动被卸载和端口被打开时重复. close 函数没有返回值, 因为它不被认为会失败.

18.2.2. 数据流

write 函数被用户在有数据发送给硬件时调用. 首先 tty 核心接收到调用, 接着它传递数据到 tty 驱动的 write 函数. tty 核心还告知 tty 驱动要发送的数据大小.

有时, 因为速度和 tty 硬件的缓冲区容量, 不是所有的写程序要求的字符可以在调用写函数时发送. 这个写函数应当返回能够发送给硬件的字符数(或者在以后时间可排队发送), 因此用户程序可以检查是否所有的数据真正写入. 这种检查在用户空间非常容易完成, 比一个内核驱动站着睡眠直到所有的请求数据能够被发送. 如果任何错误发生在 wirte 调用期间, 一个负的错误值应当被返回代替被写入的字节数.

write 函数可从中断上下文和用户上下文中被调用. 知道这一点是重要的, 因为 tty 驱动不应当调用任何可能当它在中断上下文中睡眠的函数. 这些包括任何可能调用调度的函数, 例如普通的函数 copy_from_user, kmalloc, 和 printk. 如果你确实想睡眠, 确信去首先检查是否驱动在中断上下文, 通过调用 calling_in_interrupt.

这个例子 tiny tty 驱动没有连接到任何真实的硬件, 因此它的写函数简单地将要写的什么数据记录到内核调试日志. 它使用下面的代码做这个:

```
static int tiny_write(struct tty_struct *tty, const unsigned char *buffer, int count)
{
        struct tiny_serial *tiny = tty->driver_data;
        int i;
        int retval = -EINVAL;
        if (!tiny)
                return -ENODEV;

        down(&tiny->sem);
        if (!tiny->open_count)
                /* port was not opened */
                goto exit;

        /* fake sending the data out a hardware port by
        * writing it to the kernel debug log.
        */
        printk(KERN_DEBUG "%s - ", __FUNCTION__);
        for (i = 0; i < count; ++i)
                printk("%02x ", buffer[i]);
        printk("\n");

exit:
        up(&tiny->sem);
        return retval;

}
```

当 tty 子系统自己需要发送数据到 tty 设备之外, write 函数被调用. 如果 tty 驱动在 tty_struct 中没有实现 put_char 函数, 这会发生. 在这种情况下, tty 核心用一个数据大小为 1 来使用 write 函数回调. 这普遍发生在 tty 核心想转换一个新行字符为一个换行和新行字符. 这里的最大的问题是 tty 驱动的 write 函数必须不返回 0 对于这类的调用. 这意味着驱动必须写那个数据的字节到设备, 因为调用者(tty 核心) 不缓冲数据和在之后的时间重试. 因为 write 函数不能知道是否它在被调用来替代 put_char, 即便只有一个字节的数据被发送, 尽力实现 write 函数以至于它一直至少在返回前写一个字节. 许多当前的 USB-到-串口的 tty 驱动没有遵照这个规则, 并且因此, 一些终端类型不能正确工作当连接到它们时.

write_room 函数被调用当 tty 核心想知道多少空间在写缓冲中 tty 驱动可用. 这个数字时时改变随着字符清空写缓冲以及调用写函数时, 添加字符到这个缓冲.

```
static int tiny_write_room(struct tty_struct *tty)
{
        struct tiny_serial *tiny = tty->driver_data;
        int room = -EINVAL;

        if (!tiny)
                return -ENODEV;

        down(&tiny->sem);
        if (!tiny->open_count)
        {
                /* port was not opened */
                goto exit;
        }
        /* calculate how much room is left in the device */
        room = 255;
exit:
        up(&tiny->sem);
        return room;
}
```

18.2.3. 其他缓冲函数

一个工作的 tty 驱动不需要在 tty_driver 结构中的 chars_in_buffer 函数, 但是它被推荐. 这个函数被调用当 tty 核心想知道多少字符仍然保留在 tty 驱动的写缓冲中要被发送. 如果驱动能够存储字符在它发送它们到硬件之前, 它应当实现这个函数为了 tty 核心能够知道是否所有的驱动中的数据已经流出.

3 个 tty_driver 结构中的函数回调可以用来刷新任何驱动保留的数据. 它们不被要求实现, 但是推荐如果 tty 驱动能够缓冲数据在它发送给硬件之前. 前 2 个函数回调称为 flush_chars 和 wait_until_sent. 这些函数被调用当 tty 核心使用 put_char 函数回调已发送了许多字符给 tty 驱动. flush_chars 函数回调被调用当 tty 核心要 tty 驱动启动发送这些字符到硬件, 如果它尚未启动. 这个函数被允许在所有的数据发送给硬件之前返回. wait_until_sent 函数回调以非常相同的发生工作; 但是它必须等待直到所有的字符在返回到 tty 核心前被发送, 或者知道超时值到时. 如果这个传递给 wait_until_sent 函数回调的超时值设为 0, 函数应当等待直到它完成这个操作.

剩下的数据刷新函数回调是 flush_buffer. 它被 tty 核心调用当 tty 驱动要刷新所有的仍然在它的写缓冲的数据. 任何保留在缓冲中的数据被丢失并且没发送给设备.

18.2.4. 无 read 函数?

只使用这些函数, tiny_tty 驱动可被注册, 可打开一个设备节点, 数据被写入设备, 关闭设备节点, 以驱动注销和从内核中卸载. 但是 tty 核心和 tty_driver 结构没有提供一个 read 函数; 换句话说, 没有函数调用存在来从驱动到 tty 核心获取数据.

替代一个传统的 read 函数, tty 驱动负责发送任何从硬件收到的数据到 tty 核心. tty 核心缓冲数据直到它被用户请求. 因为 tty 核心提供的缓冲逻辑, 对每个 tty 驱动不必要实现它自己的缓冲逻辑. tty 核心通知 tty 驱动当一个用户要驱动停止和开始发送数据, 但是如果内部的 tty 缓冲满, 没有这样的通知发生.

tty 核心缓冲由 tty 驱动接收到的数据, 在一个称为 struct tty_flip_buffer 的结构中. 一个 flip 缓冲是一个结构包含 2 个主要数据数组. 从 tty 设备接收到的数据被存储于第一个数组. 当这个数组满, 任何等待数据的用户被通知数据可以读. 当用户从这个数组读数据, 任何新到的数据被存储在第 2 个数组. 当那个数组被读空, 数据再次刷新给用户, 并且驱动开始填充第 1 个数组. 本质上, 被接收的数据 "flips" 从一个缓冲到另一个, 期望不会溢出它们 2 个. 为试图阻止数据丢失, 一个 tty 驱动可以监视到来的数组多大, 并且, 如果它添满, 及时告知 tty 驱动在这个时刻刷新缓冲, 而不是等待下一个可用的机会.

struct tty_flip_buffer 结构的细节对 tty 驱动没有关系, 只有一个例外, 可用的计数. 这个变量包含多少字节当前留在缓冲里可用来接收数据. 如果这个值等于值 TTY_FLIPBUF_SIZE, 这个 flip 缓冲需要被刷新到用户, 使用一个对 tty_flip_buffer_push 的调用. 这展示在下面的代码:

```
for (i = 0; i < data_size; ++i)
{
        if (tty->flip.count >= TTY_FLIPBUF_SIZE)
                tty_flip_buffer_push(tty);
        tty_insert_flip_char(tty, data[i], TTY_NORMAL);
}
tty_flip_buffer_push(tty);
```

从 tty 驱动接收来的要发送给用户的字符被添加到 flip 缓冲, 使用对 tty_insert_flip_char 的调用. 这个函数的第一个参数是数据应当保存入的 struct tty_struct, 第 2 个参数是要保存的字符, 第 3 个参数是任何应当为这个字符设置的标志. 这个标志值应当设为 TTY_NORMAL 如果这个是一个正常的被接收的字符. 如果这是一个特殊类型的指示错误接收数据的字符, 它应当设为 TTY_BREAK, TTY_PARITY, 或者 TTY_OVERRUN, 取决于错误.

为了"推"数据给用户, 进行一个对 tty_flip_buffer_push 的调用. 这个函数应当也被调用如果 flip 缓冲将要溢出, 如同在这个例子中展示的. 因此无论何时数据被加到 flip 缓冲, 或者当 flip 缓冲满, tty 驱动必须调用 tty_flip_buffer_push. 如果 tty 驱动可高速接收数据, tty->low_latency 标志应当设置, 它是对 tty_flip_buffer_pus 的调

用被立刻执行当调用时. 否则, tty_flip_buffer_push 调用会调度它自己来将数据推出缓冲, 在之后近期的一个时间点.

18.3. TTY 线路设置

当一个用户要改变一个 tty 设备的线路设置或者获取当前线路设置, 他调用一个许多的不同 termios 用户空间库函数或者直接对这个 tty 设备的节点调用 ioctl. tty 核心转换这 2 种接口为许多不同的 tty 驱动函数回调和 ioctl 调用.

18.3.1. set_termios 函数

大部分 termios 用户空间函数被库转换为一个对驱动节点的 ioctl 调用. 大量的不同的 tty ioctl 调用接着被 tty 核心转换为一个对 tty 驱动的单个 set_termios 函数调用. set_termios 调用需要决定哪个线路设置它被请求来改变, 接着在 tty 设备中做这些改变. tty 驱动必须能够解码所有的在 termios 结构中的不同设置并且响应任何需要的改变. 这是一个复杂的任务, 因为所有的线路设置以很多的方式被包装进 termios 结构.

一个 set_termios 函数应当做的第一件事情是决定任何事情是否真的需要改变. 这可使用下面的代码完成:

```
unsigned int cflag;
cflag = tty->termios->c_cflag;
/* check that they really want us to change something */
if (old_termios)
{
        if ((cflag == old_termios->c_cflag) &&
                        (RELEVANT_IFLAG(tty->termios->c_iflag) ==
RELEVANT_IFLAG(old_termios->c_iflag))) {
                printk(KERN_DEBUG " - nothing to change...\n");
                return;
        }
}
```

RELEVANT_IFLAG 宏定义为:

```
#define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

而且用在屏蔽掉 cflags 变量的重要位. 接着这个和原来的值比较, 并且看是否它们不同. 如果不, 什么不改变, 因此我们返回. 注意 old_termios 变量是第一个被检查来看是否它指向一个有效的结构, 在它被存取之前. 这是需要的, 因为有时这个变量被设为 NULL. 试图存取一个 NULL 指针成员会导致内核崩溃.

为查看需要的字节大小, CSIZE 位掩码可用来从 cflag 变量区分出正确的位. 如果这个大小无法知道, 习惯上确实是 8 个数据位. 这个可如下实现:

```
/* get the byte size */
switch (cflag & CSIZE)
{
```

```
case CS5:
        printk(KERN_DEBUG " - data bits = 5\n");
        break;
case CS6:
        printk(KERN_DEBUG " - data bits = 6\n");
        break;
case CS7:
        printk(KERN_DEBUG " - data bits = 7\n");
        break;
default:
case CS8:
        printk(KERN_DEBUG " - data bits = 8\n");
        break;
}
```

为决定需要的奇偶值, PARENB 位掩码可对 cflag 变量检查来告知是否任何奇偶要被设置. 如果这样, PARODD 位掩码可用来决定是否奇偶应当是奇或者偶. 这个的一个实现是:

```
/* determine the parity */
if (cflag & PARENB)
        if (cflag & PARODD)
                printk(KERN_DEBUG " - parity = odd\n");
        else
                printk(KERN_DEBUG " - parity = even\n");
else
        printk(KERN_DEBUG " - parity = none\n");
```

请求的停止位也可使用 CSTOPB 位掩码从 cflag 变量中来知道. 一个实现是:

```
/* figure out the stop bits requested */
if (cflag & CSTOPB)
        printk(KERN_DEBUG " - stop bits = 2\n");
else
        printk(KERN_DEBUG " - stop bits = 1\n");
```

有 2 个基本的流控类型: 硬件和软件. 为确定是否用户要求硬件流控, CRTSCTS 位掩码用来对 cflag 变量检查. 它的一个例子是:

```
/* figure out the hardware flow control settings */
if (cflag & CRTSCTS)
        printk(KERN_DEBUG " - RTS/CTS is enabled\n");
else
        printk(KERN_DEBUG " - RTS/CTS is disabled\n");
```

确定软件流控的不同模式和不同的起停字符是有些复杂:

```
/* determine software flow control */
/* if we are implementing XON/XOFF, set the start and

 * stop character in the device */
if (I_IXOFF(tty) || I_IXON(tty))
{
```

```
        unsigned char stop_char = STOP_CHAR(tty);
        unsigned char start_char = START_CHAR(tty);

        /* if we are implementing INBOUND XON/XOFF */
        if (I_IXOFF(tty))
                printk(KERN_DEBUG " - INBOUND XON/XOFF is enabled, "
                       "XON = %2x, XOFF = %2x", start_char, stop_char);
        else
                printk(KERN_DEBUG" - INBOUND XON/XOFF is disabled");

        /* if we are implementing OUTBOUND XON/XOFF */
        if (I_IXON(tty))
                printk(KERN_DEBUG" - OUTBOUND XON/XOFF is enabled, "
                       "XON = %2x, XOFF = %2x", start_char, stop_char);
        else
                printk(KERN_DEBUG" - OUTBOUND XON/XOFF is disabled");
}
```

最后，波特率需要确定. tty 核心提供了一个函数，tty_get_baud_rate，来帮助做这个. 这个函数返回一个整型数指示请求的波特率给特定的 tty 设备.

```
/* get the baud rate wanted */
printk(KERN_DEBUG " - baud rate = %d", tty_get_baud_rate(tty));
```

现在 tty 驱动已经确定了所有的不同的线路设置，它可以基于这些值正确设置硬件.

18.3.2. tiocmget 和 tiocmset

在 2.4 和老的内核，常常有许多 tty ioctl 调用来获得和设置不同的控制线路设置. 这些被常量 TIOCMGET，TIOCMBIS，TIOCMBIC，和 TIOCMSET 表示. TIOCMGET 用来获得内核的线路设置值，并且对于 2.6 内核，这个 ioctl 调用已经被转换为一个 tty 驱动回调函数，称为 tiocmget. 其他的 3 个 ioctls 已经被简化并且现在用单个的 tty 驱动回调函数所代表，称为 tiocmset.

tty 驱动中的 iocmget 函数被 tty 核心所调用，当核心需要知道当前的特定 tty 设备的控制线的物理值. 这常常用来获取一个串口的 DTR 和 RTS 线的值. 如果 tty 驱动不能直接读串口的 MSR 或者 MCR 寄存器，因为硬件不允许这样，一个它们的拷贝应当在本地保持. 许多 USB-到-串口 驱动必须实现这类的"影子"变量. 这是这个函数能如何被实现，任何一个本地的这些值的拷贝被保存:

```
static int tiny_tiocmget(struct tty_struct *tty, struct file *file)
{
        struct tiny_serial *tiny = tty->driver_ data;
        unsigned int result = 0;
        unsigned int msr = tiny->msr;
        unsigned int mcr = tiny->mcr;
        result = ((mcr & MCR_DTR)  ? TIOCM_DTR  : 0) |        /* DTR is set */
                 ((mcr & MCR_RTS)  ? TIOCM_RTS  : 0) |        /* RTS is set */
                 ((mcr & MCR_LOOP)  ? TIOCM_LOOP : 0) |       /* LOOP is set */
                 ((msr & MSR_CTS)  ? TIOCM_CTS  : 0) |        /* CTS is set */
```

```
                ((msr & MSR_CD)  ? TIOCM_CAR  : 0) |         /* Carrier detect is set*/
                ((msr & MSR_RI)  ? TIOCM_RI  : 0) |          /* Ring Indicator is set */
                ((msr & MSR_DSR)  ? TIOCM_DSR  : 0);         /* DSR is set */
        return result;
}
```

在 tty 驱动中的 tiocmset 函数被 tty 核心调用，当核心要设置一个特定 tty 设备的控制线值. tty 核心告知 tty 驱动设置什么值和清理什么，通过传递它们用 2 个变量: set 和 clear. 这些变量包含一个应当改变的线路设置的位掩码. 一个 ioctl 调用从不请求驱动既设置又清理一个特殊的位在同一时间，因此先发生什么操作没有关系. 这是一个例子，关于这个函数如何能够由一个 tty 驱动实现:

```
static int tiny_tiocmset(struct tty_struct *tty, struct file *file, unsigned int set ,
unsigned int clear)
{
        struct tiny_serial *tiny = tty->driver_data;
        unsigned int mcr = tiny->mcr;

        if (set & TIOCM_RTS)
                mcr |= MCR_RTS;
        if (set & TIOCM_DTR)
                mcr |= MCR_RTS;

        if (clear & TIOCM_RTS)
                mcr &= ~MCR_RTS;
        if (clear & TIOCM_DTR)
                mcr &= ~MCR_RTS;

        /* set the new MCR value in the device */
        tiny->mcr = mcr;
        return 0;

}
```

18.4. ioctls 函数

在 struct tty_driver 中的 ioctl 函数被 tty 核心调用当 ioctl(2) 被在设备节点上调用. 如果这个 tty 驱动不知道如何处理传递给它的 ioctl 值，它应当返回 -ENOIOCTLCMD 来试图让 tty 核心实现一个通用的调用版本.

2.6 内核定义了大约 70 个不同的 tty ioctls，可被用来发送给一个 tty 驱动. 大部分的 tty 驱动不处理它们全部，但是只有一个小的更普通的子集. 这是一个更通用的 tty ioctls 列表，它们的含义，以及如何实现它们:

TIOCSERGETLSR

获得这个 tty 设备的线路状态寄存器(LSR)的值.

TIOCGSERIAL

获得串口线信息. 调用者可以潜在地从 tty 设备获得许多串口线路信息, 在这个调用中一次全部. 一些程序(例如 setserial 和 dip) 调用这个函数来确保波特率被正确设置, 以及来获得通常的关于驱动控制的设备类型信息. 调用者传递一个指向一个大的 serial_struct 结构的指针, 这个结构应当由 tty 驱动填充正确的值. 这是一个如何实现这个的例子:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int cmd,
unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCGSERIAL)
        {
                struct serial_struct tmp;
                if (!arg)
                        return -EFAULT;
                memset(&tmp, 0, sizeof(tmp));
                tmp.type  = tiny->serial.type;
                tmp.line  = tiny->serial.line;
                tmp.port  = tiny->serial.port;
                tmp.irq  = tiny->serial.irq;
                tmp.flags  = ASYNC_SKIP_TEST | ASYNC_AUTO_IRQ;

                tmp.xmit_fifo_size = tiny->serial.xmit_fifo_size;
                tmp.baud_base = tiny->serial.baud_base;
                tmp.close_delay = 5*HZ;
                tmp.closing_wait = 30*HZ;
                tmp.custom_divisor = tiny->serial.custom_divisor;
                tmp.hub6 = tiny->serial.hub6;
                tmp.io_type = tiny->serial.io_type;
                if (copy_to_user((void __user *)arg, &tmp, sizeof(tmp)))
                        return -EFAULT;
                return 0;
        }
        return -ENOIOCTLCMD;
}
```

TIOCSSERIAL

设置串口线路信息. 这是 IOCGSERIAL 的反面, 并且允许用户一次全部设置 tty 设备的串口线状态. 一个指向 struct serial_struct 的指针被传递给这个调用, 填满这个 tty 设备应当被设置的数据. 如果这个 tty 驱动没有实现这个调用, 大部分程序仍然正确工作.

TIOCMIWAIT

等待 MSR 改变. 用户在非寻常的情况下请求这个 ioctl, 它想在内核中睡眠直到这个 tty 设备的 MSR 寄存器发生某些事情. arg 参数包含用户在等待的事件类型. 这通常用来等待直到一个状态线变化, 指示有更多的数据发送给设备.

当实现这个 ioctl 时要小心，并且不要使用 interruptible_sleep_on 调用，因为它是不安全的(有很多不好的竞争条件涉及它). 相反，一个 wait_queue 应当用来避免这个问题. 这是一个如何实现这个 ioctl 的例子:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int cmd,
unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCMIWAIT)
        {

                DECLARE_WAITQUEUE(wait, current);
                struct async_icount cnow;
                struct async_icount cprev;
                cprev = tiny->icount;
                while (1) {

                        add_wait_queue(&tiny->wait, &wait);
                        set_current_state(TASK_INTERRUPTIBLE);
                        schedule();
                            /* see if a signal woke us up */
                        remove_wait_queue(&tiny->wait, &wait);
                        if (signal_pending(current))
                                return -ERESTARTSYS;
                        cnow = tiny->icount;
                        if (cnow.rng == cprev.rng && cnow.dsr == cprev.dsr &&
                               cnow.dcd == cprev.dcd && cnow.cts == cprev.cts)
                                return -EIO;            /* no change => error */
                        if (((arg & TIOCM_RNG) && (cnow.rng != cprev.rng)) ||
                            ((arg & TIOCM_DSR) && (cnow.dsr != cprev.dsr)) ||
                            ((arg & TIOCM_CD) && (cnow.dcd != cprev.dcd)) ||
                            ((arg & TIOCM_CTS) && (cnow.cts != cprev.cts)) ) {
                                return 0;
                        }
                        cprev = cnow;
                }
        }
        return -ENOIOCTLCMD;
}
```

在 tty 驱动的代码中能知道 MSR 寄存器改变的某些地方，下面的代码行必须调用以便这个代码能正常工作:

```
wake_up_interruptible(&tp->wait);
```

TIOCGICOUNT

获得中断计数. 当用户要知道已经产生多少串口线中断时调用. 如果驱动有一个中断处理，它应当定义一个内部计数器结构来跟踪这些统计和递增适当的计数器，每次这个函数被内核运行时.

这个 ioctl 调用传递内核一个指向结构 serial_icounter_struct 的指针，它应当被 tty 驱动填充. 这个调用常常和之前的 IOCMIWAIT ioctl 调用结合使用. 如果 tty 驱动跟踪所有的这些中断在驱动操作时，实现这个调用的代码会非常简单:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int cmd,
unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCGICOUNT)
        {
                struct async_icount cnow = tiny->icount;
                struct serial_icounter_struct icount;
                icount.cts = cnow.cts;
                icount.dsr = cnow.dsr;
                icount.rng = cnow.rng;
                icount.dcd = cnow.dcd;
                icount.rx = cnow.rx;
                icount.tx = cnow.tx;
                icount.frame = cnow.frame;
                icount.overrun = cnow.overrun;
                icount.parity = cnow.parity;
                icount.brk = cnow.brk;
                icount.buf_overrun = cnow.buf_overrun;
                if (copy_to_user((void __user *)arg, &icount, sizeof(icount)))
                        return -EFAULT;
                return 0;
        }
        return -ENOIOCTLCMD;
}
```

18.5. TTY 设备的 proc 和 sysfs 处理

tty 核心提供一个非常容易的方式给任何 tty 驱动来维护一个文件在 /proc/tty/driver 目录中. 如果驱动定义 read_proc 或者 write_proc 函数，这个文件被创建. 接着，任何在这个文件上的读或写调用被发送给这个驱动. 这些函数的格式只象标准的 /proc 文件处理函数.

作为一个例子，由一个简单的 read_proc tty 回调实现，只是打印出当前注册的端口号:

```
static int tiny_read_proc(char *page, char **start, off_t off, int count,
                          int *eof, void *data)
{
        struct tiny_serial *tiny;
        off_t begin = 0;
        int length = 0;
        int i;

        length += sprintf(page, "tinyserinfo:1.0 driver:%s\n", DRIVER_VERSION);
        for (i = 0; i < TINY_TTY_MINORS && length < PAGE_SIZE; ++i) {
                tiny = tiny_table[i];
                if (tiny == NULL)
```

```
                continue;
            length += sprintf(page+length, "%d\n", i);
            if ((length + begin) > (off + count))
                goto done;

            if ((length + begin) < off) {
                begin += length;
                length = 0;

            }
        }
        *eof = 1;
done:
        if (off >= (length + begin))

            return 0;
        *start = page + (off-begin);
        return (count < begin+length-off) ? count : begin + length-off;
}
```

tty 核心处理所有的 sysfs 目录和设备创建，当 tty 驱动被注册时，或者当单个 tty 设备被创建时，依赖在 struct tty_driver 中的 TTY_DRIVER_NO_DEVFS 标志. 单个目录一直包含 dev 文件，它允许用户空间工具来决定分配给设备的主次号. 它还包含一个 device 和 driver 符号连接，如果一个指向有效的 struct device 的指针被传递给读 tty_register_device 的调用. 除了这 3 个文件，对单个 tty 驱动不可能在这个位置创建新的 sysfs 文件. 这个会可能在将来的内核发行中改变.

18.6. tty_driver 结构的细节

tty_driver 结构用来注册一个 tty 驱动到 tty 核心. 这是结构中所有不同的成员的列表和如何被 tty 核心使用:

struct module *owner;

　　这个驱动的模块拥有者.

int magic;

　　给这个结构的"魔术"值. 应当一直设为 TTY_DRIVER_MAGIC. 在 alloc_tty_driver 函数中被初始化.

const char *driver_name;

　　驱动的名子，用在 /proc/tty 和 sysfs.

const char *name;

　　驱动的节点名.

int name_base;

使用的起始数字，当创建设备名子时. 当内核创建分配给这个 tty 驱动的一个特定 tty 设备的字符串表示是使用.

short major;

驱动的主编号

short minor_start;

驱动的开始次编号. 这常常设为 name_base 的相同值. 典型地，这个值设为 0.

short num;

分配给这个驱动的次编号个数. 如果整个主编号范围被驱动使用了，这个值应当设为 255. 这个变量在 alloc_tty_driver 函数中初始化.

short type;
short subtype;

描述什么类型的 tty 驱动在注册到 tty 核心. subtype 的值依赖于 type. type 成员可能是:

TTY_DRIVER_TYPE_SYSTEM

由 tty 子系统内部使用来记住它在处理一个内部 tty 驱动. subtype 应当设为 SYSTEM_TYPE_TTY，SYSTEM_TYEP_CONSOLE，SYSTEM_TYPE_SYSCONS，或者 SYSTEM_TYPE_SYSPTMX. 这个类型不应当被任何"正常" tty 驱动使用.

TTY_DRIVER_TYPE_CONSOLE

仅被控制台驱动使用.

TTY_DRIVER_TYPE_SERIAL

被任何串行类型驱动使用. subtype 应当设为 SERIAL_TYPE_NORMAL 或者 SERIAL_TYPE_CALLOUT，根据你的驱动是什么类型. 这是 type 成员的其中一个最普遍的设置.

TTY_DRIVER_TYPE_PTY

被伪控制台接口(pty)使用. subtype 需要被设置为 PTY_TYPE_MASTER 或者 PTY_TYPE_SLAVE.

struct termios init_termios;

当创建设备时的初始化 struct termios 值.

int flags;

驱动标志, 如同本章前面描述的.

struct proc_dir_entry *proc_entry;

这个驱动的 /proc 入口结构. 它由 tty 核心创建如果驱动实现了 write_proc 或者 read_proc 函数. 这个成员不应当由 tty 驱动自己设置.

struct tty_driver *other;

指向一个 tty 从驱动. 这只被 pty 驱动使用, 并且不应当被其他的 tty 驱动使用.

void *driver_state;

tty 驱动的内部状态. 应当只被 pty 驱动使用.

struct tty_driver *next;
struct tty_driver *prev;

连接变量. 这些变量被 tty 核心使用来连接所有的不同 tty 驱动, 并且不应当被任何 tty 驱动碰.

18.7. tty_operaions 结构的细节

tty_operations 结构包含所有的函数回调, 可以被一个 tty 驱动设置和被 tty 核心调用. 当前, 所有包含在这个结构中的的函数指针也在 tty_driver 结构中, 但是会很快被只有一个这个结构的实例来替代.

int (*open)(struct tty_struct * tty, struct file * filp);

open 函数.

void (*close)(struct tty_struct * tty, struct file * filp);

close 函数.

int (*write)(struct tty_struct * tty, const unsigned char *buf, int count);

write 函数.

void (*put_char)(struct tty_struct *tty, unsigned char ch);

单字节写函数. 这个函数被 tty 核心调用当单个字节被写入设备. 如果一个 tty 驱动没有定义这个函数, write 函数被调用来替代, 当 tty 核心想发送一个单个字节.

```
void (*flush_chars)(struct tty_struct *tty);
void (*wait_until_sent)(struct tty_struct *tty, int timeout);
```

刷新数据到硬件的函数.

```
int (*write_room)(struct tty_struct *tty);
```

指示多少缓冲空闲的函数.

```
int (*chars_in_buffer)(struct tty_struct *tty);
```

指示多少缓冲满数据的函数.

```
int (*ioctl)(struct tty_struct *tty, struct file * file, unsigned int cmd,
unsigned long arg);
```

ioctl 函数. 这个函数被 tty 核心调用, 当 ioctl(2)在设备节点上被调用时.

```
void (*set_termios)(struct tty_struct *tty, struct termios * old);
```

set_termios 函数. 这个函数被 tty 核心调用, 当设备的 termios 设置已被改变时.

```
void (*throttle)(struct tty_struct * tty);
void (*unthrottle)(struct tty_struct * tty);
void (*stop)(struct tty_struct *tty);
void (*start)(struct tty_struct *tty);
```

数据抑制函数. 这些函数用来帮助控制 tty 核心的输入缓存. 这个抑制函数被调用当 tty 核心的输入缓冲满. tty 驱动应当试图通知设备不应当发送字符给它. unthrottle 函数被调用当 tty 核心的输入缓冲已被清空, 并且它现在可以接收更多数据. tty 驱动应当接着通知设备可以接收数据. stop 和 start 函数非常象 throttle 和 unthrottle 函数, 但是它们表示 tty 驱动应当停止发送数据给设备以及以后恢复发送数据.

```
void (*hangup)(struct tty_struct *tty);
```

挂起函数. 这个函数被调用当 tty 驱动应当挂起 tty 设备. 任何需要做的特殊的硬件操作应当在此时发生.

```
void (*break_ctl)(struct tty_struct *tty, int state);
```

线路中断控制函数. 这个函数被调用当这个 tty 驱动要打开或关闭线路的 BREAK 状态在 RS-232 端口上. 如果状态设为 -1, BREAK 状态应当打开. 如果状态设为 0, BREAK 状态应当关闭. 如果这个函数由 tty 驱动实现, tty 核心将处理 TCSBRK, TCSBRKP, TIOCSBRK, 和 TIOCCBRK ioctl. 否则, 这些 ioctls 被发送给驱动 ioctl 函数.

void (*flush_buffer)(struct tty_struct *tty);

刷新缓冲和丢失任何剩下的数据.

void (*set_ldisc)(struct tty_struct *tty);

设置线路规程的函数. 这个函数被调用当 tty 核心已改变这个 tty 驱动的线路规程. 这个函数通常不用并且不应当被一个驱动定义.

void (*send_xchar)(struct tty_struct *tty, char ch);

发送 X-类型 字符 的函数. 这个函数用来发送一个高优先级 XON 或者 XOFF 字符给 tty 设备. 要被发送的字符在 ch 变量中指定.

int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data);
int (*write_proc)(struct file *file, const char *buffer, unsigned long count, void *data);

/proc 读和写函数.

int (*tiocmget)(struct tty_struct *tty, struct file *file);

获得当前的特定 tty 设备的线路设置. 如果从 tty 设备成功获取到, 应当返回这个值给调用者.

int (*tiocmset)(struct tty_struct *tty, struct file *file, unsigned int set, unsigned int clear);

设置当前的特定 tty 设备的线路设置. set 和 clear 包含了去设置或者清除的不同的线路设置.

18.8. tty_struct 结构的细节

tty_struct 变量被 tty 核心用来保持当前的特定 tty 端口的状态. 几乎它的所有的朋友都只被 tty 核心使用, 有几个例外. 一个 tty 驱动可以使用的成员在此描述:

unsigned long flags;

tty 设备的当前状态. 这是一个位段变量, 并且通过下面的宏定义存取:

TTY_THROTTLED

当驱动以及有抑制函数被调用. 不应当被一个 tty 驱动设置, 只有 tty 核心.

TTY_IO_ERROR

由驱动设置当它不想任何数据被读出或写入驱动. 如果一个用户程序试图做这个, 它接收一个 -EIO 错误从内核中. 这常常在设备被关闭时设置.

TTY_OTHER_CLOSED

只由 pty 驱动使用来通知, 当端口已经被关闭.

TTY_EXCLUSIVE

由 tty 核心设置来指示一个端口在独占模式并且只能一次由一个用户存取.

TTY_DEBUG

内核中任何地方都不用.

TTY_DO_WRITE_WAKEUP

如果被设置, 线路规程的 write_wakeup 函数被允许来被调用. 常常在 tty_driver 调用 wake_up_interruptible 函数的同一时间被调用.

TTY_PUSH

只被缺省的 tty 线路规程内部使用.

TTY_CLOSING

tty 核心用来跟踪是否一个端口在那个时刻及时处于关闭过程.

TTY_DONT_FLIP

被缺省的 tty 线路规程用来通知 tty 核心, 它不应当改变 flip 缓冲, 当它被置位.

TTY_HW_COOK_OUT

如果被一个 tty 驱动设置, 它通知线路规程应当"烹调"发送给它的输出. 如果它没有设置, 线路规程成块拷贝驱动的输出; 否则, 它不得不为线路改变将单个发送的字节逐个求值. 这个标志应当通常不被 tty 驱动设置.

TTY_HW_COOK_IN

几乎和设置在驱动中的 flag 变量中的 TTY_DRIVER_REAL_RAW 标志一致. 这个标志通常应当不被 tty 驱动设置.

TTY_PTY_LOCK

pty 驱动用来加锁和解锁一个端口.

TTY_NO_WRITE_SPLIT

如果设置, tty 核心不将对 tty 驱动的写分成正常大小的块. 这个值不应当用来阻止对 tty 端口通过发送大量数据到端口的 DoS 攻击,

struct tty_flip_buffer flip;

给 tty 设备的 flip 缓冲.

struct tty_ldisc ldisc;

给 tty 设备的线路规程.

wait_queue_head_t write_wait;

给 tty 写函数的 wait_queue. 一个 tty 驱动应当唤醒它,当它可以接收更多数据时.

struct termios *termios;

指向 tty 设备的当前 termios 设置的指针.

unsigned char stopped:1;

指示是否 tty 设备被停止. tty 驱动可以设置这个值.

unsigned char hw_stopped:1;

指示是否 tty 设备的已经被停止. tty 驱动可以设置这个值.

unsigned char low_latency:1;

指示是否 tty 设备是一个低反应周期的设备, 能够高速接收数据. tty 驱动可以设置这个值.

unsigned char closing:1;

指示是否 tty 设备在关闭端口当中. tty 驱动可以设置这个值.

struct tty_driver driver;

当前控制这个 tty 设备的 tty_driver 结构.

void *driver_data;

指针, tty_driver 可以用来存储对于 tty 驱动本地的数据. 这个变量不被 tty 核心修改.

18.9. 快速参考

本节提供了对本章介绍的概念的参考. 它还解释了每个 tty 驱动需要包含的头文件的角色. 在 tty_driver 和 tty_device 结构中的成员变量的列表, 但是, 在这里不重复.

#include <linux/tty_driver.h>

头文件, 包含 struct tty_driver 的定义和声明一些在这个结构中的不同的标志.

#include <linux/tty.h>

头文件, 包含 tty_struct 结构的定义和几个不同的宏定义来易于存取 struct termios 的成员的单个值. 它还含有 tty 驱动核心的函数声明.

#include <linux/tty_flip.h>

头文件, 包含几个 tty flip 缓冲内联函数, 使得易于操作 flip 缓冲结构.

#include <asm/termios.h>

头文件, 包含 struct termio 的定义, 用于内核所建立的特定硬件平台.

struct tty_driver *alloc_tty_driver(int lines);

函数, 创建一个 struct tty_driver, 可之后传递给 tty_register_driver 和 tty_unregister_driver 函数.

void put_tty_driver(struct tty_driver *driver);

函数, 清理尚未成功注册到 tty 内核的 struct tty_driver 结构.

void tty_set_operations(struct tty_driver *driver, struct tty_operations *op);

函数, 初始化 struct tty_driver 的函数回调. 有必要在 tty_register_driver 可被调用前调用.

int tty_register_driver(struct tty_driver *driver);
int tty_unregister_driver(struct tty_driver *driver);

函数, 从 tty 核心注册和注销一个 tty 驱动.

void tty_register_device(struct tty_driver *driver, unsigned minor, struct device *device);
void tty_unregister_device(struct tty_driver *driver, unsigned minor);

对 tty 核心注册和注销一个单个 tty 设备的函数.

void tty_insert_flip_char(struct tty_struct *tty, unsigned char ch, char flag);

插入字符到 tty 设备的要被用户读的 flip 缓冲的函数.

TTY_NORMAL
TTY_BREAK
TTY_FRAME
TTY_PARITY
TTY_OVERRUN

flag 参数的不同值，用在 tty_insert_flip_char 函数.

int tty_get_baud_rate(struct tty_struct *tty);

函数，获取当前为特定 tty 设备设置的波特率.

void tty_flip_buffer_push(struct tty_struct *tty);

函数，将当前 flip 缓冲中的数据推给用户.

tty_std_termios

变量，使用一套通用的缺省线路设置来初始化一个 termios 结构.